Pyxis UIE Sequencers Functional Spec

This document describes the overall architecture of the UIE Sequencers, and the specific capabilities and instruction-set architecture of each of the different sequencers

Contents

1 MICROCODE PROGRAMMING RULES

Hardware imposes the following restrictions on microcode programming, listed here for reference:

1.1 AMEM ADDRESS ALIGNMENT RULES

1. AMEM addresses have “partition” (16-byte) granularity. For 8-bit data types, AMEM accesses (reads or writes) may have any partition alignment. But for 16-bit data types, each AMEM access must be aligned on an even partition (32-byte) boundary.
2. When reading compressed weights (and compressed-weight metadata including the scaling factors and codebooks) every access must be aligned on a 128-byte boundary.
3. Indirection tables (used for indirect reads and writes) must be aligned starting on a 128-byte boundary.

1.2 INPUT TENSOR SIZE RULES (FOR THE GRID)

1. For 3x3 convolutions, the number of input rows is indirectly constrained to satisfy the multiple-of-8 output rows rule (section 1.3 below). In practice, this means that for conv2D, 3x3, stride=1, the number of input rows must = N*8 + 2, inclusive of any padding. For conv2D, 3x3, stride=2, the number of input rows must = N*16 + 1, inclusive of any padding.
2. For 1x1 conv / linear layers, every trip must have a perfect multiple of 8 input channels using 8-bit weights, and a perfect multiple of 4 input channels using 16-bit weights, as seen at the Grid. If necessary, the weights and input tensor data must be padded up to these multiples.

   When mixing 8-bit and 16-bit weights for different input channels in the same trip, the input channels should be grouped into contiguous sets of the same type, with the respective multiples. Note that compressed weights appear as 8-bit weights for this rule.
3. For MatMul (data x data), when the Horizontal matrix (first Matrix operand) uses FP8 values, the inner dimension of the matrix multiplication must be a perfect multiple of 8 (at the Grid), and when the horizontal matrix uses FP16 values, the inner dimension must be a perfect multiple of 4 (at the Grid).

   When mixing FP8/FP16 values for the Horizontal matrix, they should be grouped into contiguous rows of the same type. If necessary, the input matrices must be padded up to these multiples.

Padding may be done by adjusting the input tensor sizes with pad data stored in AMEM, or in microcode by generating extra zeros beyond what is stored in AMEM. The “post final” feature of the AMEM read sequencers can be used to generate the extra pad rows (see sections 5.3 and 7.3)

1.3 Output Tensor-size rules (for the Grid)

1. For 3x3 convolutions: every trip must have a perfect multiple of 8 output data-rows computed in the grid, and if necessary, the input data must be padded to produce this perfect multiple of 8 output rows.

   This is due to interaction between active accumulation, split accumulation and grid writeback, all sharing the accumulator read ports, so that each of these respective operations always accesses all 8 accumulator slots

2. For 3x3 convolutions: every trip must have output channels which are a perfect multiple of "filters per grid-row". This means that for narrow tensors which fit 2, 4 or 8 filters (output channels) on each grid-row, the total output channels must be a multiple of this number. If necessary, the weights must be padded with fake output channels.

3. For 1x1 conv / linear layers: every trip must have a perfect multiple of 8 output channels, and if necessary, the weights must be padded with fake output channels to produce this perfect multiple of 8 output channels.

   This is due to the 8 "virtual grid rows" per (physical) grid-row of the LLC Grid.

4. For MatMul (data x data): the output matrix (or tile) must have a perfect multiple of 8 output data-rows, and if necessary, the horizontal input data must be padded with extra input rows to produce this perfect multiple of 8 output rows.

   This is due to the 8 "virtual grid rows" per (physical) grid-row of the LLC Grid.

If padding is necessary, this may be done by adjusting the input tensor sizes with pad data stored in AMEM, or in microcode by generating extra zeros beyond what is stored in AMEM. Extra (pad) output data rows/channels may be **discarded by microcode after offloading from the Grid**. The "post final" feature of the AMEM write sequencer can be used to discard excess rows (see section 6.3)

1.4 Weights Compression Tensor-size rules

1. When using 1x1 compressed weights to represent multiple input channels (for one output channel) in each compression block, the input channels must be padded up to at least the block size (4, 8, 16 or 32), so that every block is fully utilized.

2. When using 1x1 compressed weights to represent multiple output channels (for one input channel) in each compressed block, the output channels must be padded up to at least the block size (4, 8, 16 or 32), so that every block is fully utilized.

If padding of compressed weights is required, this must be done by adding pads within the compressed-weights blocks stored in memory.

2 UIE SEQUENCER ARCHITECTURE

The requirements for sequencing engines can be illustrated by using the AMEM Read sequencer as an example. A typical sequencing pattern has the form of nested loops. For a typical 3x3 stride1 convolution layer, the main block of nested loops iterates over groups of 8 rows (+2 overlapping rows) for each input channel. (The pattern for Matrix multiplication is similar but simpler.)

Below is pseudo-code for this example. This assumes that tensors in AMEM are indexed by (Channel, Row), to read one row per cycle. The mapping from the dimension indices to the specific AMEM addresses would be configured parameters, and are ignored here for simplicity. In the following, H_in is the input tensor height (#rows), C_in is input channel count.

```
for i0 in range (0, H_in/8):              # loop on row-groups of 8 rows
    for i1 in range (0, C_in):            # loop on input channels per row-group
        for i2 in range (0,10):           # row within group, with 2 row overlap
            READ(chan=i1, row=(i0*8)+i2)
```

Sequencers are designed to execute these nested loops efficiently, supporting 6 nested loop *iterators* within a single instruction. In graphical form, this simple sequence can be mapped as follows. Note that each sequencer micro-instruction specifies the loop controls *and* the desired operations – there are no separate instructions just for branching or loop controls.

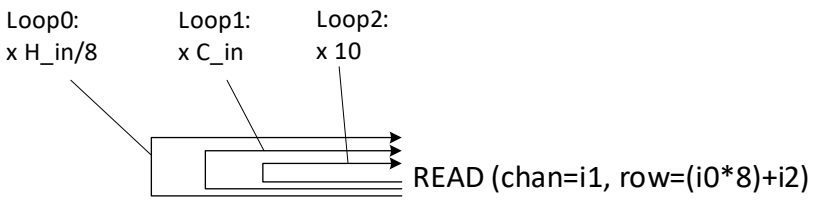


Figure 2-1: Nested Loops example

The generated datapath controls are a function of the loop iterators. The microprograms which execute each trip are deterministic and pre-calculated, with no data-dependent conditional loops.

The basic Sequencer architecture is illustrated in the figure below.

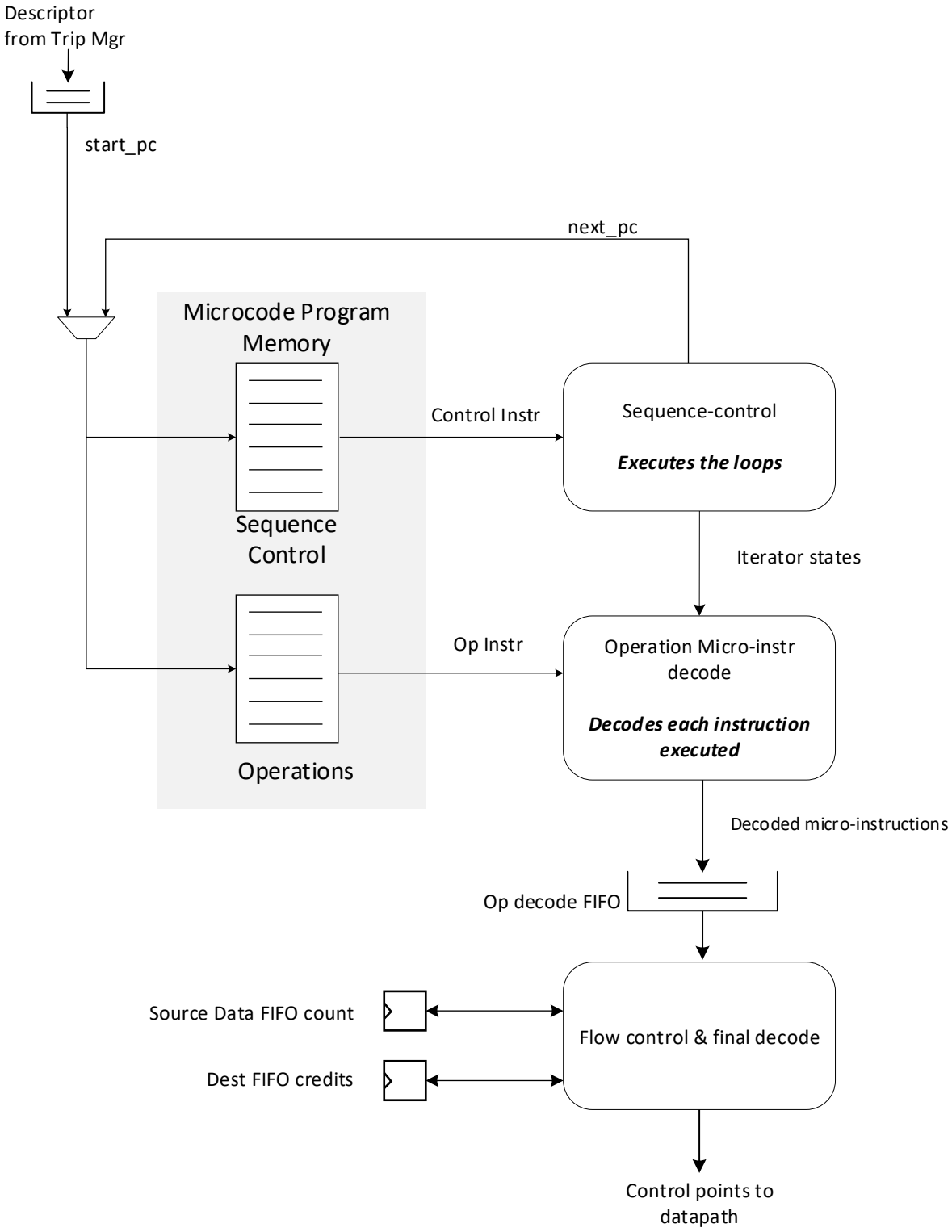


Figure 2-2: Simplified Sequencer Architecture

The microcode instructions for this engine have a *control* part and an *operations* part:

- The c*ontrol* portion of the microinstructions includes loop controls (is_EndOfLoop, LoopStartPC, NumLoops) for multiple nested loops. Each microinstruction executes for (possibly) many cycles as it iterates through the loops, and sequences of microinstructions can also form loops. The hardware iterators (total of 6) control the nested loops.
- The *operations* portion of the microinstructions provides the control-points to perform the desired operations. For example, the AMEM Read Sequencer provides the READ instructions and addresses to the memory. For the LLC Grid Exec Sequencer, the control points include the accumulator slot to read & write, and the operation (matmul, conv3x3, etc.) to execute across the grid.

The generated control points are then written to a FIFO for each executed microinstruction. These controls also indicate any resource dependencies, such as if a source FIFO needs to have data available or destination FIFOs have credits. When all dependencies are satisfied, the control FIFO is popped and the control points are validated and sent to the datapath. This can backpressure the control FIFO, which subsequently inhibits the sequencer from executing the program, and regulates the sequencer operation rate.

The sequencing control part of the microinstructions (common to all sequencers) is described in the following section. Per-sequencer operations microinstructions are detailed in the subsequent sections.

2.1 Microinstruction Format - Sequencing Part

The microinstruction sequencing part has the following general format for each microinstruction. This is common across all sequencer instances:

iter 0 spec	iter 1 spec	…	iter 5 spec	end of program

The End of Program (EOPGM) flag is set for the last instruction, and will trigger the sequencer to signal “done” when the EOPGM instruction completes.

The `iter<i> spec` (iterator specifications) defines the loop controls for each nested loop, controlled by that iterator. Each `iter<i>` specification has the following format:

Field	Size (bits)	Description
is_eol	1	Indicates this microinstruction is end of the loop for iterator *i*
sol_pc	5	Start-of-loop pc for this loop level. (Meaningful only if **is_eol**)
numloops	12	Total iterations to be executed at the loop for iterator *i*. **numloops** must be set if **is_eol==1.** **numloops** may also be set if **is_eol==0**, for intermediate instructions in a loop to inform the hardware when the instruction is in the last iteration of a loop. Encoded as N-1, so 0=1 loop iteration
numloops_final	12	Total iterations to be executed at the loop for iterator *i*, for final loop invocation as indicated by **final_iter_mask**. **numloops_final** is only meaningful if **final_iter_mask** != 0**.** **numloops_final** must be set if **is_eol==1** and **final_iter_mask** !=0. **numloops_final** may also be set if **is_eol==0** and **final_iter_mask**!=0**,** for intermediate instructions in a loop to inform the hardware when the instruction is in the last iteration of a loop. **numloops_final** is not meaningful for iterator 0, since there is no **final_iter_mask** for iterator 0. Encoded as N-1, so 0=1 loop iteration
final_iter_mask	1 .. N_{iters}-1	Mask indicates which higher-level iterators are considered to determine if the loop for iterator `i` is in its “final” invocation. Iterator `i` has `i` possible outer loops, hence `i` **final_iter_mask** bits. The **final_iter_mask** does not exist for iterator 0. If **final_iter_mask**==0, then **numloops_final** is ignored and there is no special handling for final loops for this iterator for this instruction.
<optional> post_final_enbl	1	If set, the sequencer will continue for **numloops** iterations, even if **numloops_final** would apply, but the sequencer core will also indicate these instructions are “post_final”. This may be used by the sequencer decoder to modify the instruction, such as converting to insert padding. This bit is only supported by the AMEM Read, AMEM Write, and AMEM Weights Reader sequencers

Table 2-1 : Iterator Control Specification

The UIE sequencers support **6 iterators** per sequencing instruction, allowing 6 nested loops. Iterator 0 is the outer-most loop, and each nested inner loop must use a higher-numbered iterator; this is a hardware requirement.

The iterator scheme allows multiple nested loops which may start or terminate on any instruction, without any “wasted” instructions for the loop controls. Hardware maintains a current loop count for each iterator, incremented when it reaches an instruction marked as **is_eol** (end of loop) for that iterator loop-level, and reset to zero when the **numloops** is reached. If an instruction is EOL for one loop-level, then the next program-counter (PC) is either the associated **sol_pc** (Start of Loop PC) for that iterator (if iter.count < **numloops**) else the next sequential PC. If an instruction is EOL for multiple loops, then the deepest loop (highest numbered iterator) is processed first, and if that has reached its **numloops** count, then the next deepest iterator is processed, and so forth.

2.1.1 Early loop termination with “numloops_final”

UIE Sequencers support terminating inner loops early, when one or more outer loop iterators have reached their final values. This allows for a compound iteration count without loop unrolling. For example, 3x3 convolutions may access groups of 10 data-rows with two row overlap, so each group advances by 8 rows. (See Figure 2-1.) This requires one (outer-loop) iterator counting by groups of 10, and another (inner-loop) iterator counting the 10 rows within each group. If there are a total of, say, 22 rows in a tensor, then the outer iterator would count through 3 groups (starting from row 0, 8, 16), but the last group only has 6 valid rows. We need to indicate that the inner-loop iterator should only count through 6 rows during the last value of the outer loop.

This is indicated with **numloops_final** and the **final_iter_mask** fields (per iterator). If all (outer) iterators indicated by **final_iter_mask** are in their final loop iteration, the (inner) iterator will terminate its loop at **numloops_final** rather than at **numloops**.

For the above example, the loop controls with numloops_final can be described in the following pseudo-code:

```
numloops0 = 3;          // NOTE: using “N” not “N-1” numbering here
numloops1 = 10;
numloops_final1 = 6;

for (iter0 = 0; iter0 < numloops0; iter0++) {           // 3 outer loops
   is_final0 = iter0 == numloops0 - 1;

    for (iter1 = 0; iter1 < (is_final0 ? numloops_final1 : numloops1); iter1++) {
       // inner loop 10,10,6
       … loop actions …
    }
}
```

Figure 2-3: Numloops_final execution example

Evaluation of this final-loop example:

Iter 0 count	**Iter 1 count**	**Total loop iterations**
0	0..9	10 loops
1	0..9	10 loops
2	0..5	6 loops

2.1.1.1 Cascaded numloops_final

Numloops_final may be *cascaded* from outer to inner loops through intermediate loops. This can be clarified with an example: consider ***three*** nested loops, each with 4 iterations, but designed to count a total of 58 iterations (instead of the full 4x4x4=64). The programming for this would be as follows, where the “is_final” determination cascades during evaluation from loop0 to loop1 to loop2:

```
numloops0 = 4;       // NOTE: using "N" not "N-1" numbering here
numloops1 = 4; numloops_final1 = 3;
numloops2 = 4; numloops_final2 = 2;

for (iter0 = 0; iter0<numloops0; iter0++) {
    is_final0 = iter0==numloops0 - 1;
    for (iter1 = 0; iter1 < (is_final0 ? numloops1 : numloops_final1); iter1++) {
        is_final1 = is_final0 && iter1==numloops_final-1;
        for (iter2 = 0; iter2 < (is_final1 ? numloops2 : numloops_final2); iter2++) {
            … loop actions …
        }
    }
}
```

Figure 2-4: Cascaded Numloops_final execution example

Evaluation of this cascaded final-loop example:

Iter 0 count	**Iter 1 count**	**Iter 2 count**	**Total loop iterations**
0	0..3	0..3	16 loops
1	0..3	0..3	16 loops
2	0..3	0..3	16 loops
3	0	0..3	4 loops
	1	0..3	4 loops
	2	0..**1**	2 loops – cascaded numloops_final

Note that outer loops have lower-numbered iterators, by definition. The final_iter_mask for iter<i> can only specify iterators in the range [0, i-1]. Iterator 0 can never employ **the numloops_final**.

2.1.2 Post-final execution option

Some sequencers also support executing instructions past the **numloops_final** count. This is configured with the **post_final_enbl** bit (per iterator). When this is set, the sequencer core will not terminate the final loop early, but rather signals when instructions are past the **numloops_final** count. This is used by the sequencer's instruction decoder to modify the "post_final" instructions

Post-final is supported by **AMEM Read**, **AMEM Write** and **AMEM Weights-Read** sequencers. The AMEM Read & Weights-Read sequencers convert post-final `Read*` instructions into `Read_Const(0)`. The AMEM Write sequencer converts post-final `Write` and `RMW_Add` instructions into `Discard`.

A key application is for padding out the last loop iteration to a fixed multiple (such as 10 rows, or 8 channels) even if the input tensor size is short of the multiple.

As an example, consider the 3x3 convolution with 22 input data rows discussed above. The AMEM Read sequencer can be programmed with **numloops**=10, **numloops_final**=6 and **post_final_enbl**=1 for the inner loop. This instructs the sequencer to read 6 data rows from AMEM on the final loop, and then generate pad data (zeros) for the remaining 4 rows.

For the preceding example of 3x3 convolution with 22 valid data rows, the post-final execution for AMEM Read sequencer is as shown in the following pseudo-code:

```
numloops0 = 3;       // NOTE: using "N" not "N-1" numbering here
numloops1 = 10;
numloops_final1 = 6;

for (iter0 = 0; iter0<numloops0; iter0++) {   // 3 outer loops
    is_final0 = iter0==numloops0 - 1;

    for (iter1 = 0; iter1 < numloops1; iter1++) {
        is_post_final1 = is_final0 && iter1 >= numloops_final1;

        if (is_post_final1 && iter1_post_final_enbl)
            Read_Const(0)
        else
            Read_SRAM(…)
    }
}
```

Figure 2-5: Post-final execution example

Evaluation of this post-final-loop example, for the AMEM Read Sequencer:

Iter 0 count	**Iter 1 count**	**Total loop iterations**	**Instructions executed**
0	0..9	10 loops	Read_SRAM
1	0..9	10 loops	Read_SRAM
2	0..5	6 loops	Read_SRAM
	6..9	4 loops	Read_Const(0)

2.1.2.1 Cascaded Post-Final processing

"Cascading" of numloops_final is also supported for post-final execution. For the previous example of 3 loops with 4x4x4 iterations, but terminating after 58 loops; if post_final is enabled, this still executes all 64 loops, but the first 58 would be normal execution and the last 6 loops woud be "post final".

```
numloops0 = 4;          // NOTE: using "N" not "N-1" numbering here
numloops1 = 4; numloops_final1 = 3;
numloops2 = 4; numloops_final2 = 2;

for (iter0 = 0; iter0<numloops0; iter0++) {
   is_final0 = iter0==numloops0 - 1;

   for (iter1 = 0; iter1 < numloops1; iter1++) {
      is_final1        = is_final0 && iter1==numloops_final-1; // NOT for loop exit
      is_post_final1   = is_final0 && iter1 >= numloops_final1;

      for (iter2 = 0; iter2 < numloops2; iter2++) {
         is_post_final2   = is_post_final1 ||
                            is_final1 && iter2 >= numloops_final2;

         if (is_post_final2 && iter2_post_final_enbl)
            Read_Const(0)
         else
            Read_SRAM(…)
      }
   }
}
```

Figure 2-6: Cascaded post-final execution example

Evaluation of this cascaded post-final-loop example, for AMEM Read sequencer:

Iter 0 count	**Iter 1 count**	**Iter 2 count**	**Total loop iterations**	**Instructions executed**
0	0..3	0..3	16 loops	Read_SRAM
1	0..3	0..3	16 loops	Read_SRAM
2	0..3	0..3	16 loops	Read_SRAM
3	0	0..3	4 loops	Read_SRAM
	1	0..3	4 loops	Read_SRAM
	2	0..1	2 loops	Read_SRAM
		2..3	2 loops	Read_Const(0)
	3	**0..3**	4 loops	Read_Const(0)

2.1.3 Iters_eq_zero and iters_eq_nloops vectors

The sequencer core provides two output vectors to the operations decoder, **iters_eq_zero** and **iters_eq_nloops**, indicating (for each cycle) whether each iterator is zero (first loop) or at its numloops value (last loop). This helps the sequencer decoder to generate certain control-points only on the first or last loop.

In case of using numloops_final, there are two special cases for this: (1) normally the **iters_eq_nloops** vector bit is set if **iter.count==iter.numloops_final** (and enabled for numloops_final). However, if the **post_final_enbl** is also set, then **iters_eq_nloops** bit is sent when **iter.count=iter.numloops** always.

2.1.4 Sequencer Loop Execution Algorithm

For each instruction, executed once per cycle, the sequencer reads microcode memory from the program-counter (PC) location, and executes the following loop traversal algorithm. This updates the PC and the iterators each cycle.:

```
// evaluate ending loop counts from outermost to innermost iterators
void find_ending_loop_counts() {
   cnt_eq_numloops_eff[0]                     = iter[0].count==iter[0].numloops-1;
   cnt_eq_numloops_eff_ignoring_postfinal[0]  = iter[0].count==iter[0].numloops-1;
   is_post_final = false;

   for (i = 1..MAX_ITERATOR_ID) {             // max==innermost iter
      is_final_outer[i] = (iter[i].final_iter_mask != 0); //init true if nonzero mask
      for (outer = 0..i-1) {
         if (iter[i].final_iter_mask[outer] &&
            !cnt_eq_numloops_ignoring_postfinal[outer]
         )
            is_final_outer[i] = false;
      }

      cnt_eq_numloops_eff_ignoring_postfinal[i] =
         iter[i].count==iter[i].numloops-1 ||
         (is_final_outer[i] && iter[i].count==iter[i].numloops_final-1);

      cnt_eq_numloops_eff[i] =
         iter[i].count==iter[i].numloops-1 ||
         (is_final_outer[i] && iter[i].count==iter[i].numloops_final-1 &&
          !iter[i].post_final_en);

      // report if any iterator is in "post final" state
      is_post_final |= is_final_outer[i] &&
                       iter[i].post_final_en &&
                       iter[i].count > iter[i].numloops_final-1
   }
}

// evaluate loop controls to determine next pc, from inner to outer iterators
void eval_iterator(int i) {
   is_last_loop = cnt_eq_numloops_eff[i];
   if (!iter[i].is_eol || is_last_loop) {
      // iter[i] is not EOL or has reached its loop limit; pass control up

      if (iter[i].is_eol) iter[i].count=0; // clear iterator back to zero

      if (i==0) {             // outermost iterator
         pc++;
         program_done = is_eopgm;
         return;
      } else {
         eval_iterator(i-1);     // pass control to next outer loop
      }
   }
   else   {
      // iter[i] is EOL and has not reached its loop limit -> loop back!
      iter[i].count++;
      pc = iter[i].sol_pc;
      return;
   }
}
```

```
For each microinstruction, execute {
    is_post_final = false;  // init
    program_done = false;
    find_ending_loop_counts();
    eval_iterator(MAX_ITERATOR_ID);  // max==innermost iter
}
```

Figure 2-7: Sequencer Loop Traversal Algorithm

2.2 MICROINSTRUCTION FORMAT - OPERATIONS PART

Each microinstruction has an operations part, which specifies the desired function or control-points to be asserted as a function of the sequencer PC and iterator counts. This is customized for each sequencer application. There is one Operations microcode word for each Sequencing-Control microinstruction, indexed by the same PC. Details of the operations microinstructions are listed in the following sections for each sequencer application.

3 SEQUENCER MICROARCHITECTURE

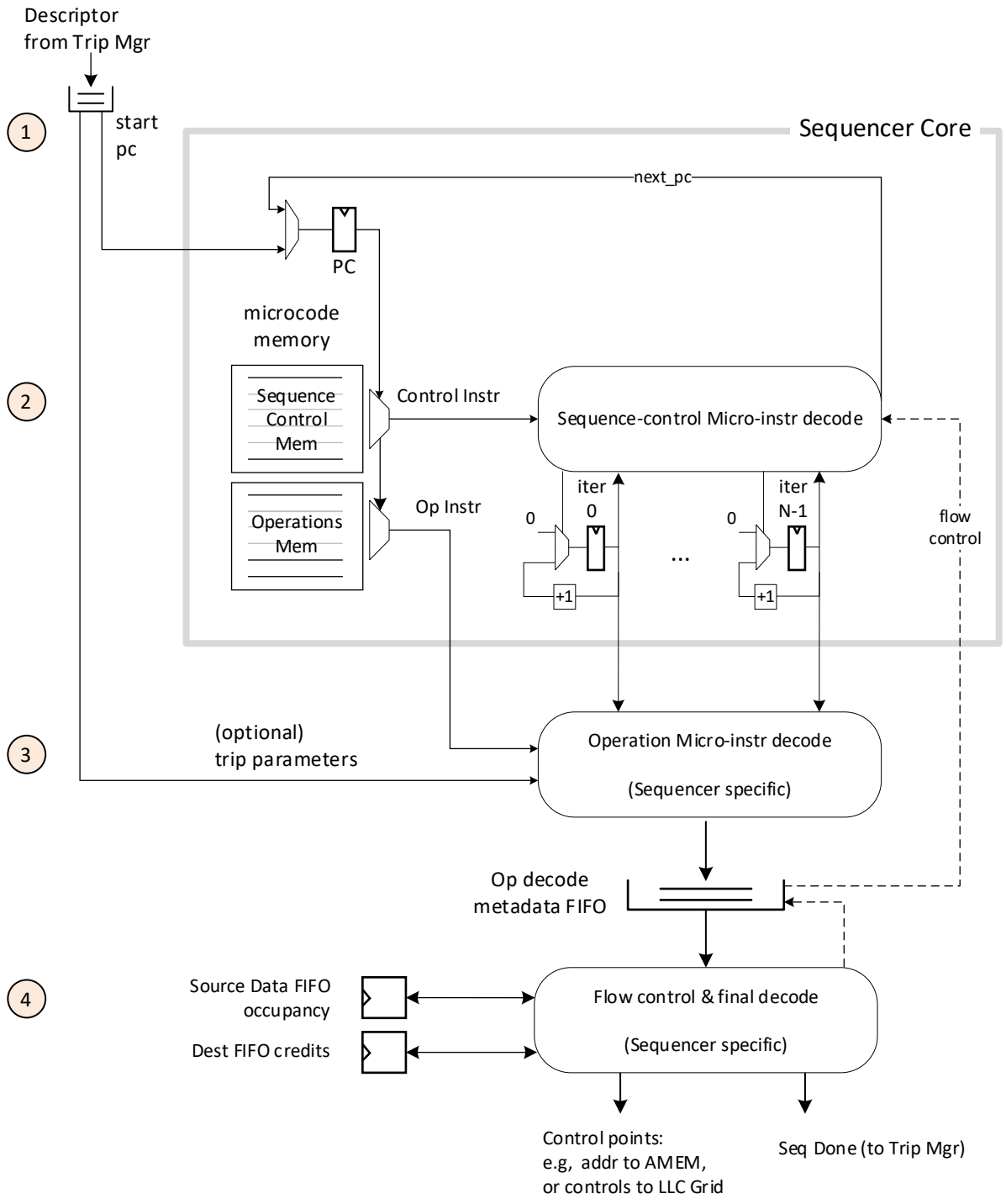


Figure 3-1: Sequencer Microarchitecture

Sequencers include the following components, as labeled in Figure 3-1:

1. Each sequencer has a shallow descriptor FIFO to receive the trip parameters, which includes the starting program counter (**StartPC**) for microcode and any sequencer-specific parameters (such as a base address for AMEM accesses). Sequencers will begin executing as soon as the trip parameters FIFO is non-empty. At the end of each sequencer program, i.e., trip, the sequencer will immediately begin executing the next trip if the parameters FIFO is again non-empty.

2. The Sequencer Core executes the microcode. This includes the **sequence-control memory** and (sequencer-specific) **operations memory**. Each cycle, the core executes one microinstruction fetched from the current program counter (PC) address. This updates the loop iterators, calculates the next PC, and forwards the iterators and operations-instruction to the instruction decoder.

   The Sequencer Core also provides vectors indicating which iterators are zero or at their "numloops" value, indicating the start or end of the loop for that iterator. This helps the sequencer program to generate some control points only on the first or last loop. For example, when operating the grid, we may want to read "zeros" instead of actual data from the accumulators for the first channel of a long accumulation. And after the last channel of the accumulation, we can kick the write-back of the accumulators.

   Sequencer core operation is subject to flow control from the decoder-output FIFO (described below). In case of flow control, the sequencer core stalls on the current PC.

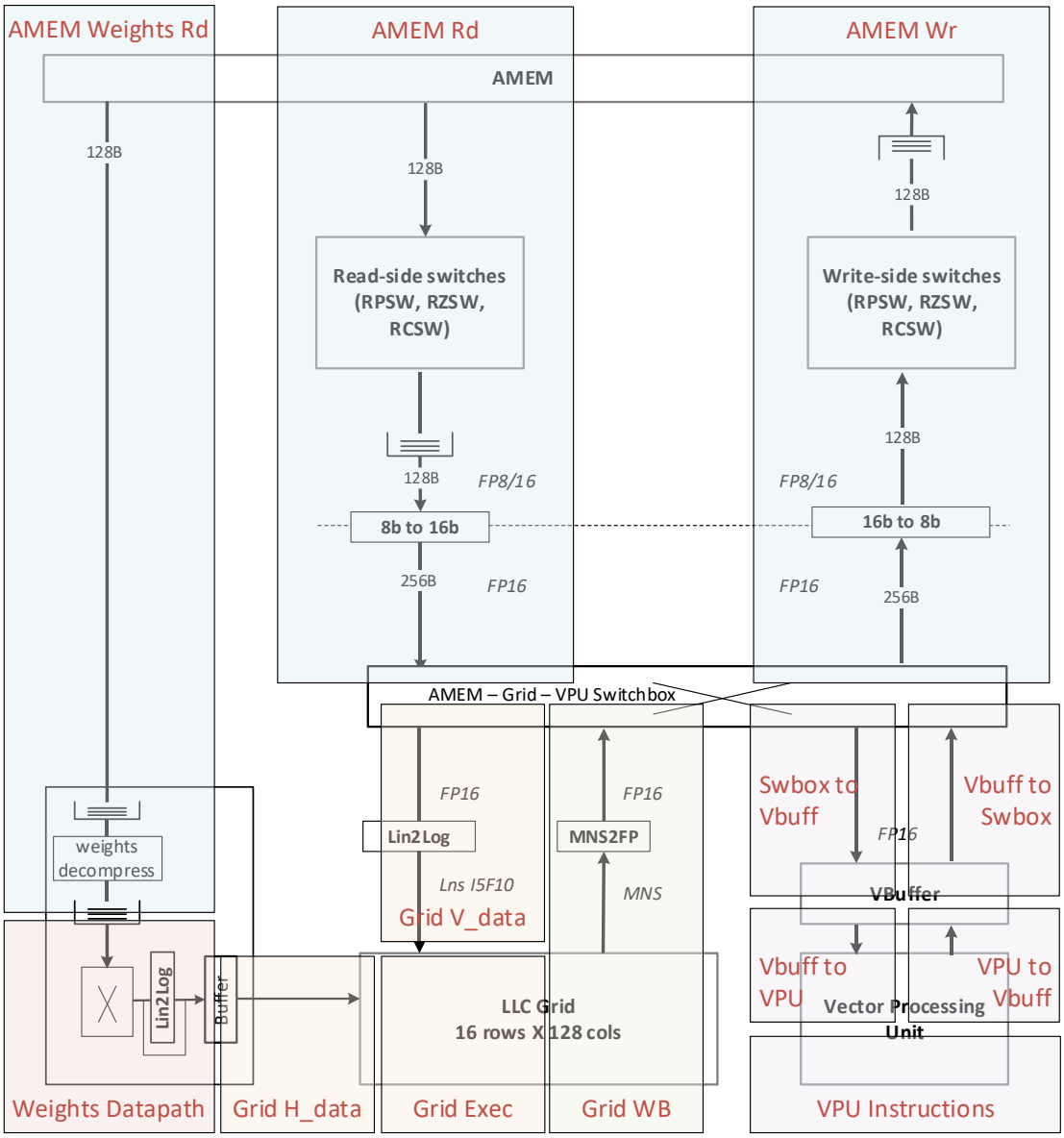
3. Sequencer-specific decoder logic interprets the operations-instruction and the iterators to produce the control points to the data path. These controls are written to a decoder-metadata FIFO. Up to this point, the operation is independent of the datapath – the sequencer is just producing a series of controls to operate the datapath.

4. The logic on the read-side of the Op decoder-metadata FIFO manages flow control in the datapath, such as whether data is available from a source FIFO, or credits are available in a destination FIFO. When the datapath dependencies are satisfied, and the decoder-metadata FIFO is not empty, the controls are sent out to the datapath, and decoder-metadata FIFO is popped. Any final decoding needed can be done here as well, and credit counters are updated.

   This is how the sequencer program adapts to the datapath operation speed: in case of flow control in the datapath, the decoder-metadata FIFO will become full, which will stall the sequencer core. In the absence of flow control, a new microinstruction can be executed every cycle.

   If this is the last instruction of the program, "sequencer done" is signaled back to the trip manager at this point.

4 SUMMARY OF UIE SEQUENCERS

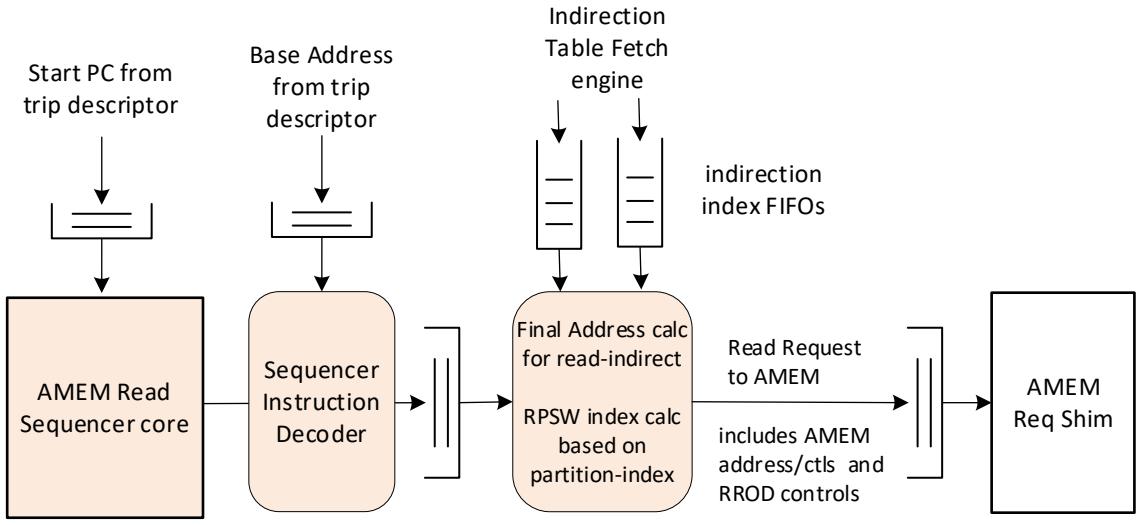


The sequencers and their domains are summarized as follows:

- 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 the Switchbox
- 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.
- 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 rest of the Weights Datapath

- Weights Datapath Write: 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.
- Top-of-Grid Control: consists of three component sequencers:
    - Grid-Horizontal-Data: (GH) coordinates movement of horizontal data from the weights buffer memories to the weights-staging buffers in the LLCs.
    - Grid-Vertical-Data (GV) coordinates movement of vertical data from the Switchbox to data-staging buffers in the LLCs.
    - Grid-Execution (GX) and 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
- Vbuffer read/write sequencers: four sequencers which control movement of data between the Switchbox and Vbuffer, and between Vbuffer and VPU.
- VPU Instruction Sequencer: Issues VPU instructions to execute VPU algorithms

5 AMEM READ SEQUENCER



The AMEM Read Sequencer provides controls to read AMEM and control the blocks in RROD portion of the UIE, including Read Partition Switch (RPSW), Read Column Switch (RCSW), FP8/16 converter and Exponent Bias converter

Each cycle, the sequencer can produce one request to read (up to) 128 *logical* tensor columns, where each logical column may be one or two bytes for 8-bit or 16-bit data types, respectively. These requests are sent to the AMEM Request Shim logic, which can convert this into one or two AMEM 128-byte read accesses (depending on data type). A FIFO decouples the sequencer and the shim logic.

5.1 AMEM READ SEQUENCER OP INSTRUCTION FORMAT

Field	Width	Description
opcd	2	0=Nop, 1=Read_Const, 2=Read_SRAM, 3=Read_SRAM_with_ReLU
data_type	3	0=Recogni FP8 1=OCP FP8 E4M3 2=OCP FP8 E5M2 3=Opaque_8bit 4=Recogni FP16 5=IEEE FP16 6=Opaque_16bit
tgt_fifo	2	selects switchbox target as one of Grid, Vbuffer or AMEM-write
amem_addr_offset	22	AMEM partition-granular starting offset address [25:4].

<table>
<tr><td></td><td></td><td>For 16-bit data types, this must be an even value, to align tensors on a 32-byte boundary</td></tr>
<tr><td>iter_stride[0..5]</td><td>6*22</td><td>Per-iterator partition-granular address “strides”. The stride is multiplied by the associated iterator value and summed into the address calculation.</td></tr>
<tr><td>indir_vld</td><td>1</td><td>If set, addresses will be computed from indices loaded into X/Y indirection FIFOs by the Indirection Table Fetch Engine (ITFE). See section 8.1 for ITFE programming.</td></tr>
<tr><td>indir_mode</td><td>2</td><td>Indirection mode: if indir_vld is set, this controls how the indirection FIFOs are used. Note: this must be programmed consistently with the indir_mode programmed in the associated AMEM Read Indirection Table Fetch Engine (ITFE). See section 8.1 for ITFE programming
<table>
<tr><th>mode</th><th>1D or 2D index?</th><th>x_index FIFO usage</th><th>Y_index FIFO usage</th></tr>
<tr><td>0</td><td>1D</td><td>16b x_index</td><td>Unused</td></tr>
<tr><td>1</td><td>1D</td><td>32b x_index</td><td>Unused</td></tr>
<tr><td>2</td><td>1D</td><td>16 lsb’s of x_index</td><td>16 msb’s of x_index</td></tr>
<tr><td>3</td><td>2D</td><td>16b x_index</td><td>16b y_index</td></tr>
</table></td></tr>
<tr><td>indir_fifo_pop_enbl</td><td>1</td><td>If set, enable pop of the active x/y indirection FIFO(s). If not set, the FIFO entries are held steady. This allows the same index values to be used for multiple AMEM reads. This is only meaningful if indir_vld is set. The FIFOs to which this applies are determined by the indir_mode setting.</td></tr>
<tr><td>rd_const_value</td><td>16</td><td>Constant value of the type defined by data_type. Only meaningful if Opcd=Read_Const

For 8-bit data types, the value is programmed in the low-order byte.</td></tr>
<tr><td>num_logical_ptns</td><td>3</td><td>number of logical partitions to read, where a logical partition is 16 tensor columns. (Each tensor column may occupy 1 or 2 bytes).

Range of values is [1..8], where codepoint 0 = all 8 partitions</td></tr>
<tr><td>start_row_pad</td><td>1</td><td>Instructs the AMEM Read to return all-zeros when the iterators specified by pad_row_iter_mask are all zero. This effectively converts a Read, Read_RELU or Read_Const opcode into Read_Const with constant value=0, when the iterator mask condition is satisfied

This is useful when the first row of a tensor (or first row of each channel) should be a pad-row (all zeros).</td></tr>
<tr><td>end_row_pad</td><td>1</td><td>Instructs the AMEM Read to return all-zeros when the iterators specified by pad_row_iter_mask are equal to their maximum value(s) as specified in the loop controls for those iterators. This effectively converts a Read, Read_RELU or Read_Const opcode into Read_Const with constant value=0, when the iterator mask condition is satisfied

This is useful when the last row of a tensor (or last row of each channel) should be a pad-row (all zeros).</td></tr>
<tr><td>pad_row_iter_mask</td><td>6</td><td>Mask of iterator(s) which control start/ending pad row. Instructs the sequencer which iterators should be zero for the start_row_pad function, and/or which iterators should have their ending value for the end_row_pad function</td></tr>
</table>

		When the mask itself is zero, start/end row pad is disabled
rpsw_config_idx_base	6	base address into the RPSW configuration table, from which the specific config index is calculated by the sequencer
rpsw_rot_enbl	1	If set, the sequencer will calculate the RPSW config index (offset) based on the effective partition-granular rotation of data, matching the rotation calculated for the AMEM address. This can index a configuration which rotates the data in the opposite direction from the AMEM rotation.
rpsw_config_iter_id	3	which iterator is used for computing the offset into the RPSW configuration table. Only the 3 LSBits of the selected iterator's count_value are used, to allow for selecting up to 8 unique configurations in a loop
rpsw_config_iter_stride	3	Multiplies the selected iterator's value by this configured stride to compute the PSW config-table offset. Only the 3 LSBits of the selected iterator's value are multiplied, to allow for striding through the config table with a stride of up to 7 config-entries per each successive iterator value. The computed index offset, modulo 8, is added to the **rpsw_config_idx_base** This may be set to zero to disable using an interator to compute the RPSW config table index offset.
rpsw_config_vld	1	RPSW configuration is valid (will be applied). If not set, the prior config is held steady. This allows a config computed in one instruction to be used in subsequent instructions
rcsw_switch_config_idx_base	6	base address into the RCSW switch configuration table, from which the specific config index is calculated by the sequencer
rcsw_switch_config_iter_id	3	which iterator is used for computing the offset into the RCSW switch configuration table. Only the 4 LSBits of the selected iterator's count_value are used, to allow for selecting up to 16 unique configurations in a loop.
rcsw_switch_config_iter_stride	4	Multiplies the selected iterator's value by this configured stride to compute the RCSW config-table offset. Only the 4 LSBits of the selected iterator's value are multiplied, to allow for striding through the config table with a stride of up to 15 config-entries per each successive iterator value. The computed index offset, modulo 16, is added to the **rcsw_switch_config_base** This may be set to zero to disable using an interator to compute the RCSW config table index offset.
rcsw_switch_config_vld	1	RCSW switch configuration is valid (will be applied). If not set, the prior config is held steady. This allows a config computed in one instruction to be used in subsequent instructions
rcsw_col_mask_config_idx	2	selects the column override "mask" configuration
rzsw_config	2	zipper-switch mode configuration. 0=NOP, 1=Unzip, 2= Zip
eb_adj	6	Exponent Bias adjustment applied during **8bit-to-16bit** conversion. Signed, 6-bit integer

5.1.1 Read_SRAM_with_ReLU operation

For the Read_SRAM_with_ReLU opcode, AMEM will apply a ReLU (Rectified Linear Unit) operation to all columns of data being read, where

```
ReLU(x) = (x==NaN) ? NaN : max(x, 0)
```

This is only supported for Recogni FP8 and Recogni FP16 data types. Applying this opcode for other data types is undefined.

5.2 AMEM READ ADDRESS CALCULATION

Each cycle the sequencer calculates the address of the starting partition in AMEM, which specifies reading up to 8 *logical* partitions from that starting address. Each logical partition is 16 tensor columns, and each tensor column may be one or two bytes. This supports reading up to 128 contiguous bytes starting at any 16-byte memory address for 8-bit data, or up to 256 contiguous bytes starting at any 32-byte memory address for 16-bit data. AMEM hardware will determine which *physical* partitions (and their memory addresses) must be read as a result.

5.2.1 Direct AMEM Address Calculation

For normal AMEM reads, the address is calculated as follows:

```
amem_address[26]   = hash_mode_from_descriptor[26]  # hashed-view bit
amem_address[25:4] = base_address_from_descriptor[25:4] +
                     amem_addr_offset[25:4]  # offset given in op instr

for i in range(6):   # per iterator
   addr_offset[25:4] = iter[i].cnt[11:0] * iter[i].stride[15:0]
   amem_address[25:4] += addr_offset[25:4]
```

Note that the base address is given in the trip descriptor, and sequencer instructions compute the offset from that base. It is legal to set the base address as zero, and supply all the address terms from the instruction – either method works. The trip-descriptor also provides the hashed-mode “view” bit (bank hashed or non-hashed), which is bit[26] of the AMEM address, i.e, an extra address bit to provide the two “views” of the AMEM address space.

The base address and strides are in units of 16 bytes, so this calculates the address of a (physical) partition. When reading 16-bit data (FP16, Int16) this address ***must*** be aligned on an even partition (32-byte) boundary.

Note: the computed address for 16b ***data*** (fp16, int16) must be an even partition address. In general, for 16b data, the base address must be an even partition address, and the strides must be an even number of partitions.

5.2.2 Indirect AMEM Address Calculation

For indirect addressing, indirection tables (stored in AMEM) provide a list of tensor index values, which must be converted to AMEM addresses. Hardware supports either 1-dimension or 2-dimension indexing. An **Indirection Table Fetch** engine is programmed to read the indirection (index) values from AMEM, and load them into "x_index" and "y_index" FIFOs feeding to the AMEM Read sequencer. (See section 8 on page 36)

There are four indirection modes supported:

- **Mode 0**: 1-D indexing with 16-bit index values supplied by the x_index FIFO.
- **Mode 1**: 1-D indexing with 32-bit index values supplied by the x_index FIFO. This may be useful when a CPU is preparing the indirection table as an array of 32-bit integers stored contiguously in memory.

  Mode 2: 1-D indexing with 32-bit index values, where the 16 LSBits are supplied by the x_index FIFO, and 16 MSBits are supplied by the y_index FIFO. In this mode, the 16 MSBits effectively overload the y_index FIFO.

  This may be useful when the VPU is preparing the indirection table, and stores each 32-bit integer values as two int16 values in separate channels in AMEM. In this scheme, the MSBits of each index would be in one (contiguous) array, and LSBits in another (contiguous) array
- **Mode 3**: 2-D indexing with 16-bit index values supplied by the x_index and y_index FIFOs.

To compute the effective AMEM address, the indices are multiplied by *strides*, and added to the tensor's computed AMEM base address (as described in section 5.2.1) as follows:

- **Iter[5].stride** is reassigned as the **X-index stride**.
- **Iter[4].stride** is reassigned as the **Y-index stride**, but only if 2D gather is specified ("Mode 3")

The reassigned iterators' strides are ***not*** used for the base address calculation in indirect mode. I.e., iterator 5 may not be used to compute the AMEM base address for read-indirect operations, since the stride value is "stolen" for the X-index stride. Similarly, Iterator 4 may not be used for calculating AMEM base address for 2D read-indirect (Mode 3) since the iterator 4 stride value is also "stolen" as the Y-index stride for indirection Mode 3.

```
amem_address[26]   = hash_mode_from_descriptor[26]  # hashed-view bit
amem_address[25:4] = base_address_from_descriptor[25:4] +
                     amem_addr_offset[25:4]  # offset given in op instr

num_base_address_iters = 4 if (is_2d_gather) else 5
for i in range (num_base_address_iters):
   addr_offset[25:4] = iter[i].cnt[11:0] * iter[i].stride[15:0]
   amem_address[25:4] += addr_offset[25:4]


# Compute the address_offset by scaling indirection indices:
if read_indirect_mode == 0:
   addr_offset_x[25 :4] = x_fifo_entry[15:0] * iter[5].stride[15:0]
```

```
elif read_indirect_mode == 1:
   addr_offset_x[25:4] = x_fifo_entry[31:0] * iter[5].stride[15:0]
elif read_indirect_mode == 2:
   addr_offset_x[25:4] = (x_fifo_entry[15:0] + y_fifo_entry[15:0]<<16) *
                         iter[5].stride[15:0]
else: # read_indirect_mode == 3 (2D)
   addr_offset_x[25:4] = x_fifo_entry[15:0] * iter[5].stride[15:0]
   addr_offset_y[25:4] = y_fifo_entry[15:0] * iter[4].stride[15:0]


# Add index offset to the computed base address:
amem_address[25:4] += addr_offset_x[25:4]
if read_indirect_mode == 3:  # 2D
   amem_address[25:4] += addr_offset_y[25:4]
```

The base address and strides are in units of 16 bytes, so this calculates the address of a (physical) partition. When reading 16bit data (FP16, Int16) the computed amem_address ***must*** be aligned on an even partition (32-byte) boundary.

5.3 AUTOMATIC PADDING FOR "POST-FINAL" READ INSTRUCTIONS.

The AMEM Read sequencer supports "post final" loop control (see section 2.1.2). This allows the sequencer program to terminate a final loop early (using **num_loops_final**) ***or*** allows the program to proceed past the **num_loops_final** value, with the "post-final" instructions flagged by the sequencer core. The post-final instructions are converted into read-padding-data operations. This can be used to complete loops of a required quantum -- e.g., 8 input channels for 1x1 convs -- when the input tensor size is not a multiple of the required quantum, by automatically adding pads for the "missing" tensor rows. When **post_final_enbl** is set, post-final iterations automatically convert Read opcodes into Read_Const with constant value = 0.

Post-final operation is progammed in the sequencer's control-word, and is specified individually for each iterator. In a typical case, only one iterator will operate as "post-final", typically an inner-loop iterator which counts through a prescribed number of rows or channels.

5.4 RPSW CONFIG TABLE INDEX CALCULATION

The Read Partition Switch (RPSW) uses a 64-entry configuration table to control the switch mux settings. The sequencer calculates the RPSW configuration table **index** for each flit, i.e., for each 8 *logical*-partitions worth of data, which is 128 bytes for 8-bit data and 256 bytes for 16-bit data types. The RPSW index is computed as:

- The base index specified with each instruction, plus
- The modulo-**8** sum of
    - The computed starting partition index of the AMEM address. This is useful to correct for unaligned memory reads – i.e., if the AMEM read address has a starting offset =***n*** (in range 0..7), the sequencer and RPSW can be jointly programmed to index an RPSW configuration to rotate the data by ***n*** partitions to re-align (un-rotate) the data.

- a sequencer index and stride, modulo 8

```
rpsw_config_index_FP8 =
   rpsw_config_base +
   (
      (rpsw_rot_enbl ? computed_partition_index : 0) +
      (iter[rpsw_config_iter_id] & 0x7) * rpsw_config_iter_stride
   ) % 8
```

This allows for tables of 8 entries processed by one looping instruction. This is a typical case for iterating over 8 possible data alignments from memory for FP8 data.

Note that setting `rpsw_config_iter_stride`= 7 effectively walks the configurations backwards (modulo 8).

Also note that for FP16 (or any 16-bit-type) data, the `computed_partition_index` in the above equation will always be one of 0, 2, 4 or 6, since 16-bit data must always be accessed from an even partition offset.

5.5 RCSW CONFIG TABLE INDEX CALCULATION

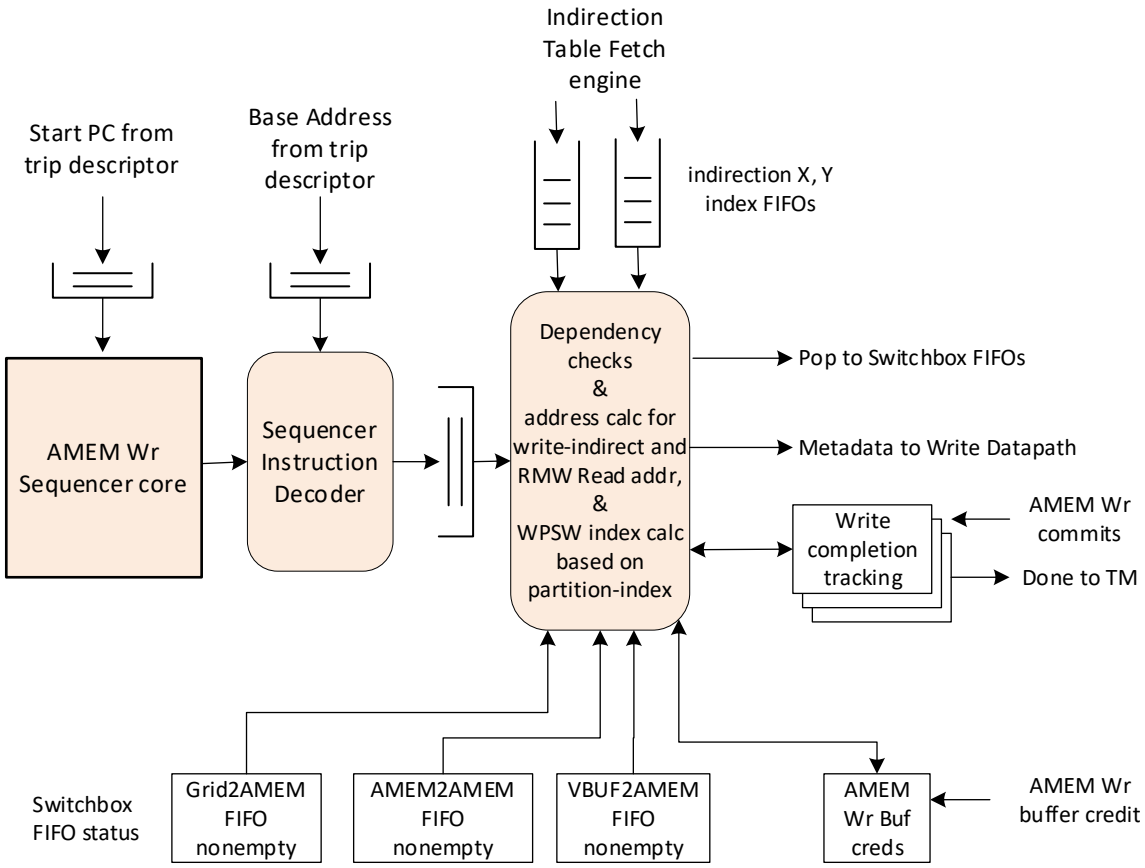
The Read Column Switch (RCSW) provides a 64-entry switch configuration table for the first two phases of the RCSW configuration, comprising the column-shifting 3:1 mux configuration and stacked-butterfly switch configuration.

The sequencer calculates an index into the RCSW switch configuration table from a base index + (iterator * stride) % 16. This allows for iterating over 16 possible switch configurations from one instruction. This is a typical case for shifting data by up to 16 columns iteratively.

```
rcsw_switch config_index =
  rcsw_switch_config_base +
  (
   (iter[rcsw_switch_config_iter_id] & 0xF) * rcsw_switch_config_iter_stride
   ) % 16
```

Note that setting `rcsw_switch_config_iter_stride`= 0xF effectively walks the configurations backwards (modulo 16) in case that is of interest.

6 AMEM WRITE SEQUENCER



The AMEM Write Sequencer provides controls and metadata to move data from the Switchbox, through the Write-datapath switching elements, and to be written to AMEM. This controls the FP16/8 converter and Exponent Bias converter, the Write Column Switch (WCSW), Write Zipper Switch (WZSW) and Write Partition Switch (WPSW), and calculates the address for AMEM.

Each cycle, the sequencer can generate control metadata for one flit (128 x FP16 tensor columns). If the data is to be written to AMEM as an 8-bit data type, these controls can be launched every cycle. If data is to be written as a 16-bit type, this will be serialized over two cycles by the FP16/8 converter, and the sequencer pops one flit from the switchbox every two cycles.

6.1 AMEM WRITE SEQUENCER OP INSTRUCTION FORMAT

Field	Width	Description
opcd	2	0=Nop, 1=Discard (pop and discard data from switchbox source FIFO) 2=Write, 3=RMW_Add (InMemAdd)
data_type	2	0=Recogni FP8

<table>
<tr><td></td><td></td><td>1=OCP FP8 E4M3
2=OCP FP8 E5M2
3=Opaque_8bit
4=Recogni FP16
5=IEEE FP16
6=Opaque_16bit</td></tr>
<tr><td>src_fifo</td><td>2</td><td>selects switchbox source FIFO from one of Grid, Vbuffer or AMEM-write</td></tr>
<tr><td>amem_addr_offset</td><td>22</td><td>AMEM partition-granular starting offset. Partition-granular address [25:4]
For 16-bit data types, this must be an even value, to align tensors on a 32-byte boundary</td></tr>
<tr><td>iter_stride[0..5]</td><td>6*22</td><td>Per-iterator partition-granular address “strides”. The stride is multiplied by the associated iterator value and summed into the address calculation.</td></tr>
<tr><td>indir_vld</td><td>1</td><td>If set, addresses will be computed from indices loaded into X/Y indirection FIFOs by the Indirection Table Fetch engine (ITFE). See section 8.1 for ITFE programming</td></tr>
<tr><td>indir_mode</td><td>2</td><td>Indirection mode: if indir_vld is set, this controls how the indirection FIFOs are used.
Note: this must be programmed consistently with the indir_mode programmed in the associated AMEM Write Indirection Table Fetch Engine (ITFE). See section 8.1 for ITFE programming
<table>
<tr><th>mode</th><th>1D or 2D index?</th><th>x_index FIFO usage</th><th>Y_index FIFO usage</th></tr>
<tr><td>0</td><td>1D</td><td>16b x_index</td><td>Unused</td></tr>
<tr><td>1</td><td>1D</td><td>32b x_index</td><td>Unused</td></tr>
<tr><td>2</td><td>1D</td><td>16 lsb’s of x_index</td><td>16 msb’s of x_index</td></tr>
<tr><td>3</td><td>2D</td><td>16b x_index</td><td>16b y_index</td></tr>
</table></td></tr>
<tr><td>indir_fifo_pop_enbl</td><td>1</td><td>If set, enable pop of the active x/y indirection FIFO(s). If not set, the FIFO entries are held steady. This allows the same index values to be used for multiple AMEM writes.
This is only meaningful if indir_vld is set. The FIFOs to which this applies are determined by the indir_mode setting</td></tr>
<tr><td>rmw_rd_addr_offset</td><td>12</td><td>For RMW_Add operation, this specifies the location of the read data, as offset from the write data. The read and write data must be located in the same AMEM bank, so this offset is within one AMEM bank. This is an unsigned 12-bit integer, so to configure a negative offset, this may be programmed as 4096-abs(offset). NOTE: nonzero RMW offsets should ONLY be used with the non-hashed “view” of AMEM addressing – see Pyxis Fspec for more details.</td></tr>
<tr><td>logical_col_offset</td><td>4</td><td>Starting logical-column offset, modulo 16, where a logical column may occupy 1 or 2 bytes for 8-bit or 16-bit data types, respectively.

If this is nonzero, this implies a partial-write, so AMEM will first read the data and merge the write-data column-by-column.</td></tr>
<tr><td>num_logical_cols</td><td>7</td><td>number of logical columns to write, where a logical column may occupy 1 or 2 bytes for 8-bit or 16-bit data types, respectively.

Range of values is [1..128], where codepoint 0 = 128 logical columns

Requirement: logical_col_offset + num_logical_cols <= 128</td></tr>
<tr><td>wpsw_config_idx_base</td><td>6</td><td>base address into the WPSW configuration table, from which the specific config index is calculated by the sequencer</td></tr>
</table>

wpsw_rot_enbl	1	If set, the sequencer will calculate the WPSW config index (offset) based on the effective partition-granular rotation of data, matching the rotation calculated for the AMEM address. This can index a configuration which rotates the data in the opposite direction from the AMEM rotation.
wpsw_config_iter_id	3	which iterator is used for computing the offset into the WPSW configuration table.
wpsw_config_iter_stride	3	Multiplies the selected iterator's value by this configured stride to compute the config-table offset. Only the 3 LSBits of the selected iterator's value are multiplied, to allow for striding through the config table with a stride of up to 7 config-entries per each successive iterator value. The computed index offset, modulo 8, is added to the **wpsw_config_idx_base** This may be set to zero to disable using an interator to compute the WPSW config table index offset.
wpsw_config_vld	1	WPSW configuration is valid (will be applied). If not set, the prior config is held steady. This allows a config computed in one instruction to be used in subsequent instructions
wcsw_switch_config_idx_base	6	base address into the WCSW switch configuration table, from which the specific config index is calculated by the sequencer
wcsw_switch_config_iter_id	3	which iterator is used for computing the offset into the WCSW switch configuration table. Only the 4 LSBits of the selected iterator's count_value are used, to allow for selecting up to 16 unique configurations in a loop.
wcsw_switch_config_iter_stride	4	Multiplies the selected iterator's value by this configured stride to compute the WCSW config-table offset. Only the 4 LSBits of the selected iterator's value are multiplied, to allow for striding through the config table with a stride of up to 15 config-entries per each successive iterator value. The computed index offset, modulo 16, is added to the **rcsw_switch_config_base** This may be set to zero to disable using an interator to compute the WCSW config table index offset.
wcsw_switch_config_vld	1	WCSW switch configuration is valid (will be applied). If not set, the prior config is held steady. This allows a config computed in one instruction to be used in subsequent instructions
wcsw_col_mask_config_idx	2	selects the column override "mask" configuration
wzsw_config	2	zipper-switch mode configuration. 0=NOP, 1=Unzip, 2= Zip
eb_adj	6	Exponent Bias adjustment applied during **16bit-to-8bit** conversion. Signed, 6-bit integer

6.2 AMEM Write Address Calculation

Each cycle the sequencer calculates the address of the starting partition in AMEM, which specifies writing up to 8 *logical* partitions from that starting address. Each logical partition is 16 tensor columns, and each tensor column may be one or two bytes. Each instruction supports writing up to 128 contiguous bytes starting at any 16-byte memory address for 8-bit data, or up to 256 contiguous bytes

starting at any 32-byte memory address for 16-bit data. AMEM hardware will determine which *physical* partitions (and their memory addresses) must be written as a result.

The details of address calculation are the same as specifed for the AMEM Read Sequencer, including direct and indirect addressing. The hash-mode is also specified with the AMEM base address provided in the descriptor. See section 5.2 for details.

6.2.1 Read-Modify-Write with Add (InMemAdd)

The AMEM Write sequencer supports **RMW_Add** which is a read–add–write combined operation (also referred to as "InMemAdd"). The read data may be located separately from the write (destination) address. To support two addresses, the microcode instruction specifies the location of the read data as an ***offset*** from the write data. The read and write data must be located in the same AMEM bank, so this offset is in units of 16K bytes. The offset is an unsigned 12-bit integer, so to configure a negative offset, this may be programmed as 4096-abs(offset).

```
amem_rmw_read_address[25:14] =  # compute bank-granular part:
   amem_write_address[25:14] +      # computed write address
   amem_rmw_addr_offset[25:14]      # offset given in op instruction

amem_rmw_read_address[13:4] = amem_write_address[13:4]
```

NOTE: nonzero RMW offsets should ONLY be used with the non-hashed "view" of AMEM addressing – see Pyxis Fspec for more details.

6.3 AUTOMATIC DISCARD FOR "POST-FINAL" WRITE INSTRUCTIONS.

The AMEM Write sequencer supports "post final" loop control (see section 2.1.2). This allows the sequencer program to terminate a final loop early (using **num_loops_final**) ***or*** allows the program to proceed past the **num_loops_final** value, with the post-final instructions flagged by the sequencer core.

The post-final instructions are converted into Write-discard operations. This can be used to complete loops of a required quantum -- e.g., a multiple of 8 output rows offloaded from the grid for MatMuls -- when the tensor size does not match those multiples, by discarding the "excess" tensor data from the switchbox. When **post_final_enbl** is set, post-final iterations automatically convert Write and RMW opcodes into Discard operations.

Post-final operation is progammed in the sequencer's control-word, and is specified individually for each iterator. In a typical case, only one iterator will operate as "post-final", typically an inner-loop iterator which counts through a prescribed number of rows or channels.

6.4 WRITES WITH ARBITRARY STARTING AND ENDING COLUMN OFFSETS

The **logical_col_offset** field allows for writes from an arbitary starting (logical) column offset from the base partition address, and the **num_logical_cols** field specifies an arbitrary number of logical columns to be written starting from that offset.

This can be used for writing partial data on arbitrary tensor column boundaries. Each logical column may occupy 1 or 2 bytes for 8-bit or 16-bit data types, respectively. For example, **logical_col_offset**=11 for 16-bit data would specify a starting byte offset of 22 from the computed partition-granular base address.

This can require a Read/Merge/Write operation in hardware, when the (physical) starting or ending columns do not fall on partition boundaries. Hardware will automatically convert the Write into a Read/Merge/Write operation when required. This will read the data word from AMEM, replace (merge) the specified columns in the data, and write the result back to AMEM.

Hardware imposes the following **rule:**

```
logical_col_offset + num_logical_cols <= 128
```

This ensures that at most 8 logical partitions can be touched by a single Write with **logical_col_offset** ≠ 0

6.5 RMW_ADD ("IN MEM ADD")

- tbd

6.5.1 Addressing Rules for RMW_Add ("In Mem Add')

- Read & Write Bank must match for non-destructive RMW_Add. In most cases this requires use of non-hashed AMEM addressing mode.
- Alternatively, the AMEM offset may be constrained to a multiple of (Number_of_AMEM_banks)**2 * AMEM_bytes_per_word = 128**2 x 128 = 2Mbytes. Two tensors located with a multiple of this offset will experience the same bank hashing, and so it is legal to execut RMW_Add in this case. However, this is so restrictive it is not expected to be used often.

6.6 WCSW CONFIG TABLE INDEX CALCULATION

The WCSW is identical to RCSW, and has an identical 64-entry configuration table to control the 3:1 mux selects and butterfly switch settings of the Column Switch block. The sequencer programming follows the same format as for RCSW (see section 5.5).

6.7 WPSW CONFIG TABLE INDEX CALCULATION

WPSW is identical to RPSW, and has an identical 64-entry configuration table to control the switch. The sequencer programming follows the same format as for RPSW (see section 5.4).

One operational difference worth noting: the PSW action required for a given data rotation will be opposite between RPSW and WPSW – for RPSW, data should be rotated *toward* partition zero to correct

for the memory alignment on read; for WPSW, the data should be rotated *away* from partition zero to set up the desired memory alignment on write.

6.8 Write Completion Tracking

The AMEM write sequencer implements write-completion tracking for the UIE. This ensures that all writes sent to AMEM have been committed before the UIE signals "trip done" to the Trip Manager, which enables relaxing Barriers so that a dependent trip can read that data from AMEM.

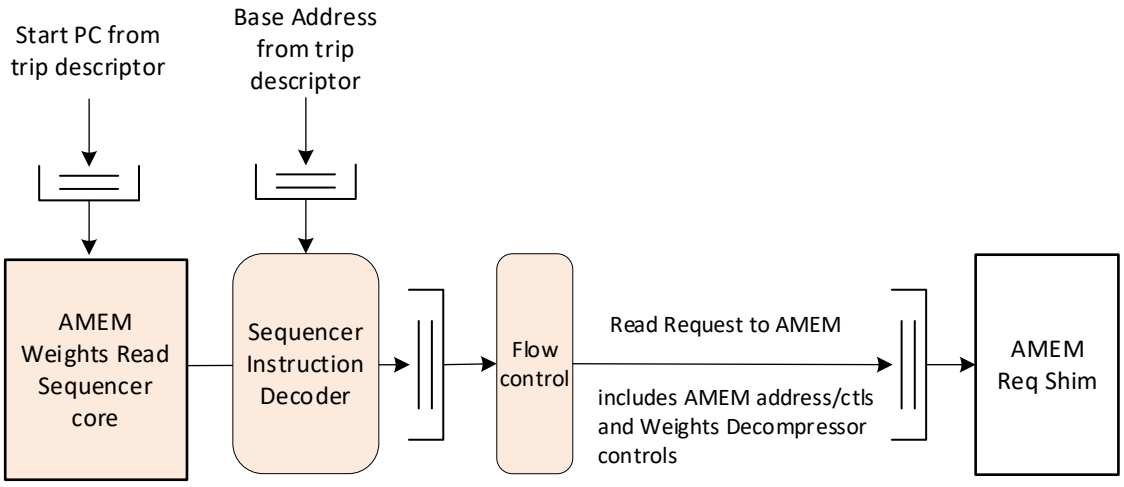
There are two "tracking IDs" and two "trackers". Each tracker provides an outstanding write counter and a state bit to record whether all writes for the trip have been issued.

Each new trip allocates the next sequential tracking ID, at the time the sequencer is started. If a tracker is not available, the sequencer launch is stalled. (This is is very unlikely, and indicates that writes have not yet completed for trip ***n*** when trying to start trip ***n+2***.) Each write sent to AMEM includes this tracking ID as metadata, and the associated tracker's counter is incremented. The last instruction for the trip (whether or not it generates an AMEM write) sets the tracker's EOPGM bit

Subsequently, AMEM returns write-commit signals for each tracking ID, and each write-commit decrements the associated tracker's counter. When the count is zero and EOPGM is set, the tracker is done. Trackers must be retired in the order they were allocated, i.e., in per-trip order. When a tracker is retired, it signals trip-done (i.e, AMEM Write sequencer's contribution to trip-done).

Write completion tracking is automatic and entirely controlled by hardware. In the normal case, it is expected that there may be one active tracker for the current trip (in the sequencer) and one tracker for the prior trip.

7 AMEM WEIGHTS READ SEQUENCER



The AMEM Weights Read Sequencer provides controls to read AMEM for the "weights" (WDP-facing) port, and metadata to control the Weights Decompressor (WDC) module.

Each cycle, the sequencer can produce one request to read 128 bytes on any 16-byte alignment, which is sent to the AMEM Request Shim which adjusts addresses for unaligned accesses. A FIFO decouples the sequencer and the shim logic.

7.1 AMEM WEIGHTS READ SEQUENCER OP INSTRUCTION FORMAT

Field	Width	Description
opcd	2	0=Nop, 1=Read_SRAM, ~~2=Read_SRAM_with_ReLU~~, 3=Read_Const
is_16bit	1	Indicates if data type is 16-bit (if set) or 8-bit (if clear).
rd_const_value	16	Constant value to apply if **Opcd**=Read_Const For 8-bit data types, the value is programmed in the low-order byte. Note that the rd_const_value may be Floating Point (FP8/FP16) or LNS (I4F3/I5F10). This is context-dependent, and the programmer's responsibility to specify the appropriate constant value.
amem_addr_offset	22	AMEM partition-granular starting offset address [25:4]. For 16-bit uncompressed data, this must be an even value, to align tensors on a 32-byte boundary. For compressed weights, codebook or scaling factor words, this must be a multiple of 8 to align to a 128-byte boundary

num_ptns	3	number of partitions to read. Range of values is [1..8], where codepoint 0 = all 8 partitions TBD: may not be required if all reads are 128B?
iter_stride[0..5]	6*16	Per-iterator **partition-granular** address "strides". The stride is multiplied by the associated iterator value and summed into the address calculation.
wdc_type	2	Indicates the weights-decompression type of the 128-byte AMEM word: 0=Uncompressed data or weights: WDC pass-through 1=Compression Codebook (occupies bytes 0-15 only) 2=Compression Scaling Factors 3=Compressed Data (4-bit weights)
wdc_block_size	2	Indicates the compression-block size for compressed 4-bit weights,. (Not meaningful for uncompressed data, scaling factors or codebook data) 0=blocksize 4 1=blocksize 8 2=blocksize 16 3=blocksize 32

For compressed weights, note that the data, codebook, and scaling-factors must be aligned on 128-byte boundaries. For uncompressed 8-bit weights or data, any partiton (16B) alignment is permitted, and for uncompressed 16-bit weights or data, any even-partition (32B) alignment is permitted.

7.2 AMEM WEIGHTS READ ADDRESS CALCULATION

This is very similar to the AMEM (data) Read Address Calculation described in section 5.2.1 above, except there is no support for indirect addressing on the AMEM weights read port:

```
amem_address[26]   = hash_mode_from_descriptor[26]  # hashed-disable view bit
amem_address[25:4] = base_address_from_descriptor[25:4] +
                     amem_addr_offset[25:4]  # offset given in op instr

for i in range(6):   # per iterator
   addr_offset[25:4] = iter[i].cnt[11:0] * iter[i].stride[15:0]
   amem_address[25:4] += addr_offset[25:4]
```

Note that the base address is given in the trip descriptor, and sequencer instructions compute the offset from that base. The trip-descriptor also provides the hashed-mode "view" bit (bank hashed or non-hashed), which is bit[26] of the AMEM address

7.3 AUTOMATIC PADDING FOR "POST-FINAL" READ INSTRUCTIONS.

The AMEM Weights Read sequencer supports "post final" loop control (see section 2.1.2). This allows the sequencer program to terminate a final loop early (using **num_loops_final**) ***or*** allows the program to proceed past the **num_loops_final** value, with the "post-final" instructions flagged by the sequencer core. The post-final instructions are converted into Read-padding operations. This can be used to complete loops of a required quantum -- e.g., 8 input channels for 1x1 convs -- when the input tensor

size is not a multiple of the required quantum, by automatically adding pads for the “missing” tensor rows. When **post_final_enbl** is set, post-final iterations automatically convert Read opcodes into Read_Const with constant value = 0.

Post-final operation is progammed in the sequencer’s control-word, and is specified individually for each iterator. In a typical case, only one iterator will operate as “post-final”, typically an inner-loop iterator which counts through a prescribed number of rows or channels.

8 AMEM Indirection Table Fetch Engine

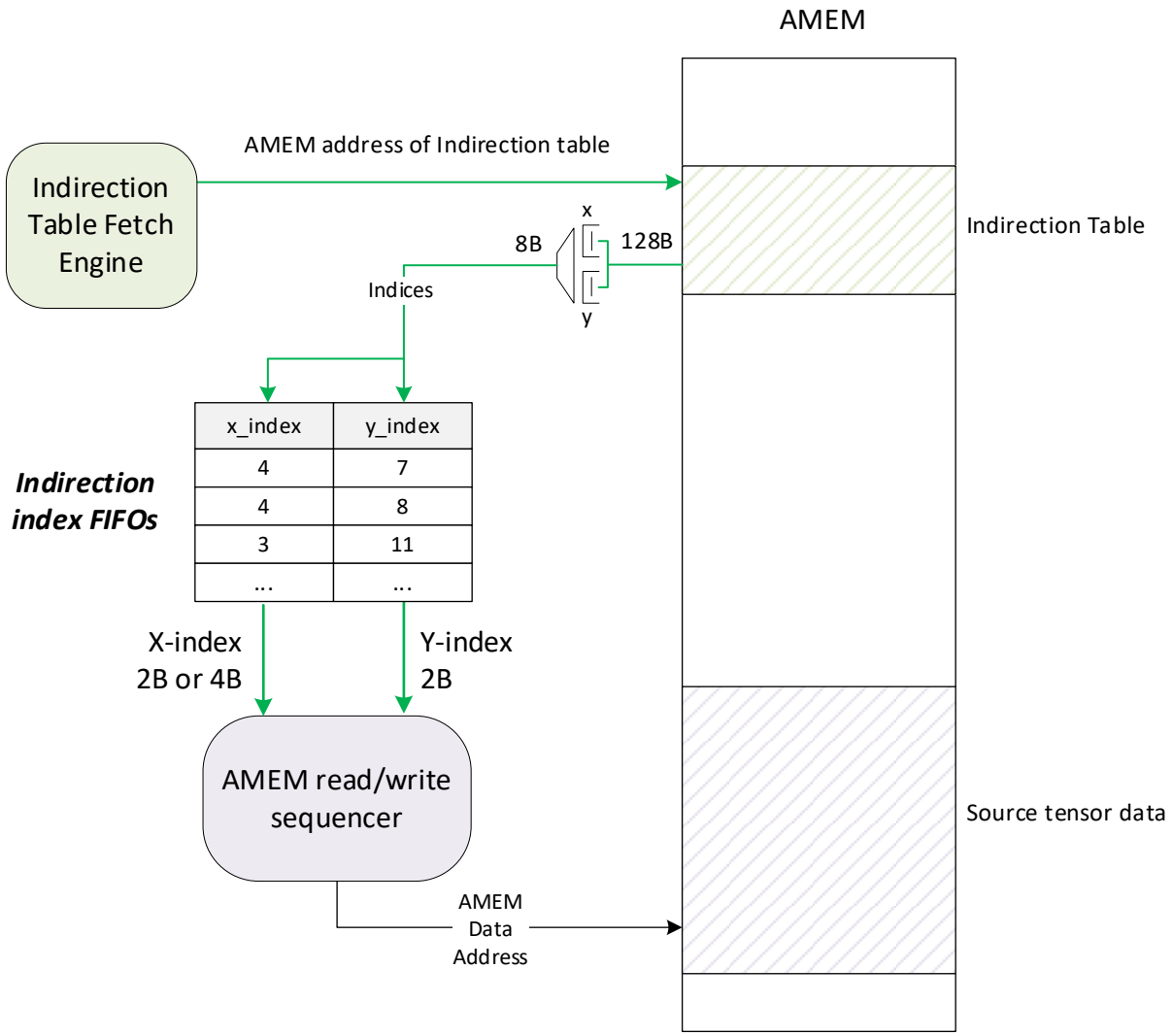


Figure 8-1: AMEM address indirection operation

The Indirection Table Fetch Engine (**ITFE**) fetches lists of indirection index values from AMEM, and supplies these values to "X-index" and "Y-index" FIFOs to be consumed by the AMEM Read/Write sequencers. This supports row-based scatter/gather operations in the UIEs with up to two dimensional indexes. There is **one ITFE for each of the AMEM Read & Write sequencers**, for each UIE. (There is no indirection support for the AMEM Weights read sequencer.) Each ITFE works closely with its associated AMEM Read or AMEM Write sequencer to supply a stream of indirection index values consumed by that sequencer.

This engine is programmed with a single command word which specifies the operation. The engine has **four** command registers, which may be programmed by software (e.g, by using the Microcode Patch Engine). For each trip, the AMEM Read/Write sequencer trip-descriptor provides the command register index, and whether indirection is enabled.

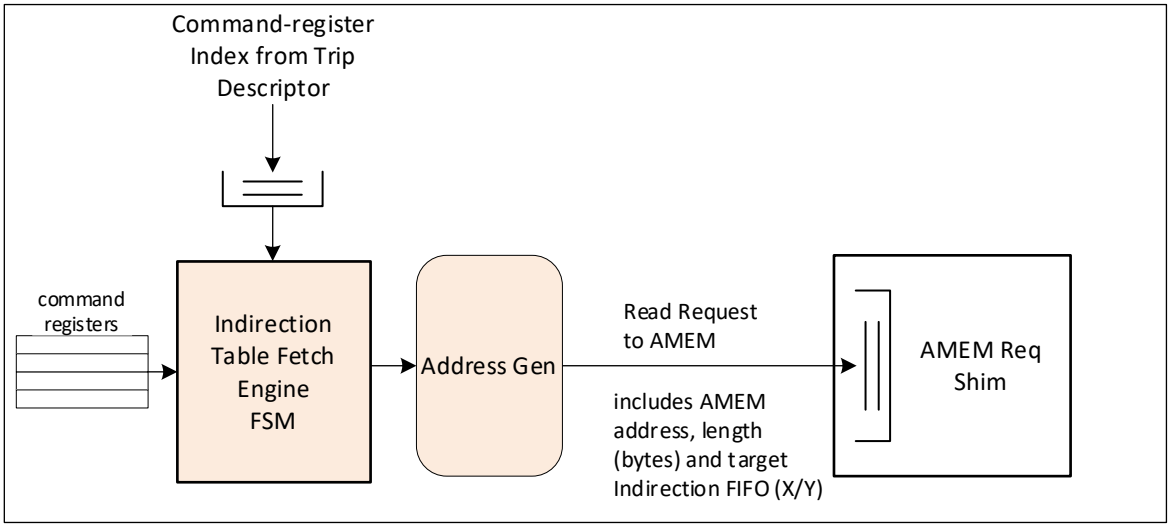


Figure 8-2: Indirection Table Fetch Engine

8.1 INDIRECTION TABLE FETCH ENGINE COMMAND FORMAT

Field	Width	Description		
indir_mode	2	**encoding**	**X_index element size**	**Y_index element size**
		0	16b	Unused
		1	32b	Unused
		2,3	16b	16
x_table_base_addr	19	AMEM 128-byte granular base address [25:7] of the indirection table. The tables must start on 128-byte aligned addresses. Y_table_base_address is only meaningful for indirection mode = 2 or 3		
y_table_base_addr	19			
bank_hash_dsbl	1	Use hashed (0) vs unhashed (1) bank addressing for reading the indirection tables from AMEM		
num_indices_per_table_m1	16	Number of index values in each table (minus 1) Encoded in "N-1" format, so codepoint 0 = 1 index, codepoint 1 = 2 indexes, ec		
repeat_count_m1	6	Repeat count for the entire table-fetch operation (minus 1). Encoded in "N-1" format, so codepoint 0 = 1 total repetition, codepoint 1 = 2 repetitions, etc		

The **indir_mode** uses the same encoding as the equivalent field in the AMEM read/write sequencer op instruction, and **must** be programmed consistently for the sequencer and the Indirection Table Fetch engine command for the same trip. This determines whether only the X index, or both X and Y index tables are used, and the per-index **element_size** (16b or 32b per index)

The **x_table_base_addr** and **y_table_base_addr** provide the starting address for X & Y index tables. These start on 128-byte boundaries. The bank_hash_dsbl indicates whether X & Y index tables are addressed with bank-hashed addressing (this is the typical case, with disable=0) or non-hashed (disable=1).

The **num_indices_per_table** specifies the number of index elements in the table(s). This is encoded in "N-1" format, and the field name is **num_indices_per_table_m1** (for "minus 1"). When both X & Y tables are used, they have the same number of elements, and are always 2 bytes/element. When only X table is used, each (X) element may be 2 bytes or 4 bytes as shown above. The engine will convert num_indices_per_table into an equivalent number of 128-byte AMEM read operations. If the num_indices_per_table * element_size (16b or 32b) is not a perfect multiple of 128 bytes, the last AMEM read may require fewer than 128 valid bytes, and hardware will fetch exactly the required number of bytes of index values to the indirection FIFOs in the read/write sequencers.

For **indir_mode**s which use both X & Y index tables, the engine will alternate between reading X and Y tables for each 128-byte read operation. The selected table (X or Y) is also provided as metadata to the AMEM read logic to steer the index values to the correct indirection FIFO in the read/write sequencer.

The **repeat_count** allows fetching the indirection table(s) multiple times per trip. This is encoded in "N-1" format, and the field name is **repeat_count_m1** (for "minus 1"). The repeat_cnt may be useful for tiling of gather or scatter operations, when the tensor data rows are very wide. Multiple tiling passes through the data may be programmed into a single trip, to read or write each successive 128-column-wide portion of the data using indirect addressing. For each tile , the associated AMEM read or write sequencer would be programmed to compute the appropriate index-to-AMEM-address calculation as a function of the tile (repeat_count) number.

Note: the total number of index values fetched by ITFE (per table) will equal **num_indices_per_table * repeat_count** and this **must** match the count of indirect memory accesses issued by the associated AMEM read or write sequencer for the same trip.

For the AMEM Read sequencer, this count includes ***Read_SRAM***, and ***Read_SRAM_with_ReLU*** instructions issued, but **excluding** those which are converted into padding (i.e., ***Read_Const***) due to **start_row_pad** or **end_row_pad** settings, or due to "post-final" read padding.

For the AMEM Write sequencer, this count includes ***Write*** and ***RMW_Add*** instructions issues, but **excluding** those which are converted into ***Discard*** due to "post-final" write suppression.

---

9 WEIGHTS DATAPATH SEQUENCER

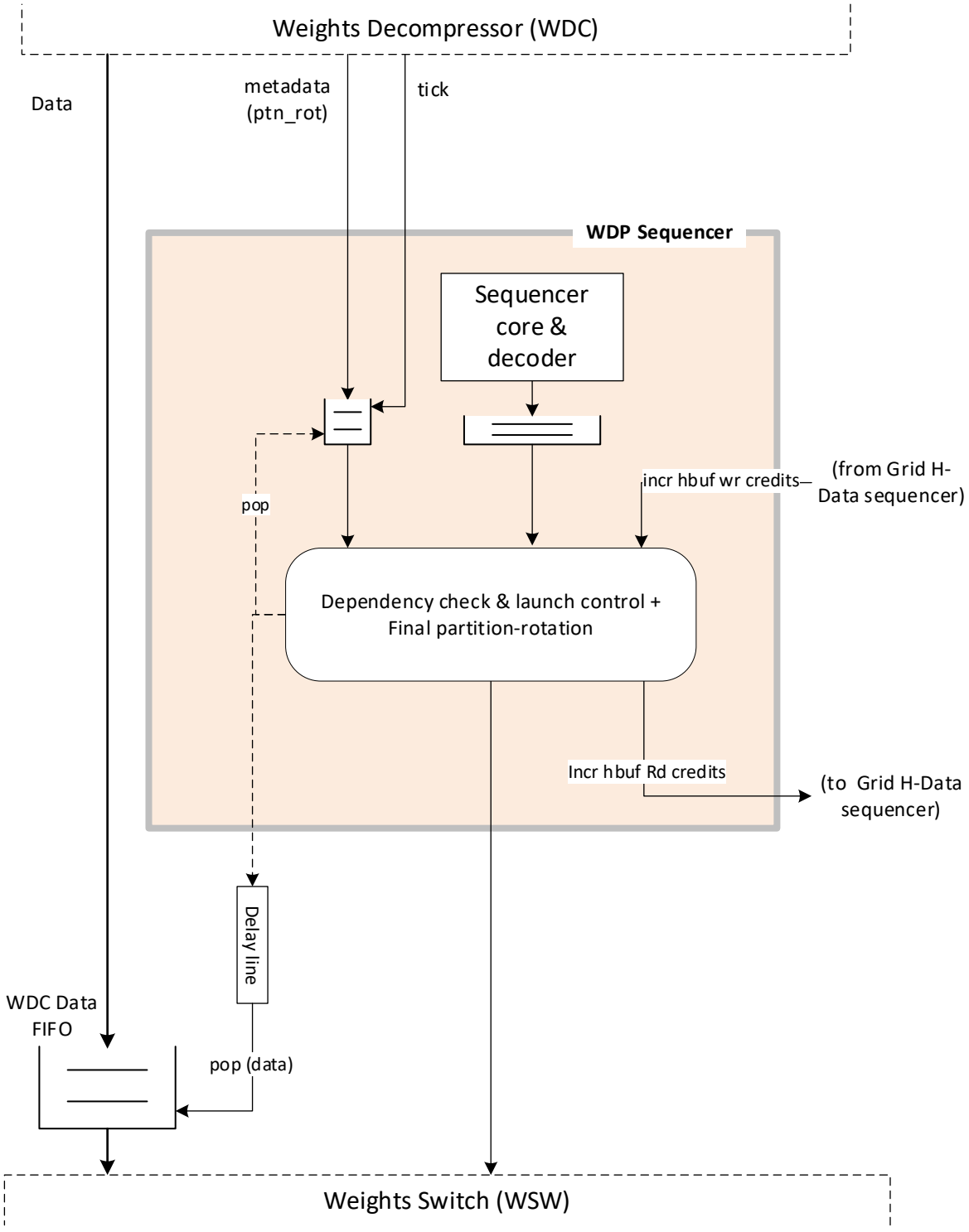


Figure 9-1: WDP Sequencer

The WDP Sequencer controls data movement from the output of the Weights decompressor to the per-grid-row horizontal data buffers (Hbufs), including all the datapath elements including the weights switch, zero-masking, linear-to-log converter and EB adjustment scaling modules

WDP Sequencer maintains credits for available space in the Hbufs, in conjunction with the Grid Horizontal-Data (GH) sequencer. The Hbufs are managed as FIFOs, but credits are incremented & decremented on a block basis, where the block size is a programmed parameter. This enables "chunky" flow control, where data is assembled in blocks in the Hbuf, but the availablilty of that data (to be consumed) is not advertised until the block is complete.

9.1 WDP SEQUENCER OP INSTRUCTION FORMAT

Field	Width	Description
opcd	1	0: Nop 1: WR_HBUF If opcd= WR_HBUF, read a word (128B) of data from the post-weights-decompression FIFO (wdc_fifo), and send it through the weights datapath to the per-grid-row Hbuf memories.
is_16bit	1	If set, the data is 16-bit (FP16 or LnsI5F10 or opaque-16-bit). If not set, data is 8-bit (FP8 or LnsI4F3 or opaque-8-bit)
Weights Switch controls		
wsw_ptn_rot_en	1	if set, use the incoming **ptn_rot[2:0]** (received from upstream) to align received data via the weights switch. If not set, the data is assumed to be 128-byte aligned
log2_ptns_per_hlane	2	1,2,4 or 8 partitions per h-lane. This also determines the number of h-lane "groups" which is 8/ptns_per_hlane. For uncompressed cases, this will be 1 partition per h-lane
hlane_iter_id	3	selects iterator to loop through h-lanes or h-lane groups, based on log2_ptns_per_hlane. Not meaningful if log2_ptns_per_lane=0.
Zero-Mask controls		
zero_mask_idx_a	2	primary zero-mask index typically sent to all h-lanes for 8-bit data or weights
zero_mask_idx_b	2	Secondary zero-mask index which may be intermixed to some h-lanes for 16-bit data or weights. Intended for FP16-transposed matmul data where two grid-rows' data are muxed onto one h-lane in the same data transfer
zero_mask_en	1	If set, zero masking is enabled using the selected zero_mask_idx
zero_mask_config_vld	1	If set, update the active zero-mask configuration with values from the preceding fields. Otherwise, the prior configuration values are retained
Lin-to-log controls		
dsbl_mapping_corr	1	0=enable lin2log conversion with mapping correction 1=disable mapping correction. This should be set if the incoming data is already in LNS format (i.e., for weights), or for a lin2log identity mapping which is used when transposing a data matrix.
eb_adj	6	EB adjustment value applied during Linear-to-log mapping in weights datapath. Signed, 6-bit integer
lin2log _config_vld	1	validates preceding lin2log and eb adjustment configs

hbuf credits management:		
hbuf_block_size	9	Number of bytes in hbuf_block (*per grid-row*), in **units of 16-byte**s. This is the block size used for "chunky" flow control with the hbuf read sequencer (which reads from hbuf logical FIFOs). 9 bits allows values up to the entire size of hbuf (256x16 bytes/grid-row). 256 is the maximum legal value. Note: LnsI5F3 data is counted as one "byte", LnsI5F10 is two "bytes"
hbuf_block_start_en	1	when all iterators indicated by hbuf_block_iter_mask are zero, instructs the sequencer to require, and deduct, hbuf_block_size credits from the hbuf (logical) FIFO write-credits. By convention, these FIFOs are incremented with block-size "chunky" credits, so it is only necessary to look for available credits at the start of a block to proceed with all writes for the entire block
hbuf_block_end_en	1	When all iterators indicated by hbuf_block_iter_mask have reached their ending loop value, instructs the sequencer that it has fully written a block of hbuf data,. When this occurs, the hbuf_base write address is advanced by the hbuf_block_size, and the hbuf_rd_credits also incremented by hbuf_block_size, advertising the amount of data is available to be read out.
hbuf_block_iter_mask	6	Mask of iterator(s) which control start/ending an hbuf block. Instructs the sequencer which iterators should be zero for the hbuf_block_start function, and/or which iterators should have their ending loop value for the hbuf_block_end function When the mask itself is zero, start/end of block is disabled
hbuf write controls		
hbuf_wr_control	2	LD_1ROW_16B - write 16 bytes/cycle to one grid-row per h-lane LD_2ROWS_8B - write 8 bytes/cycle to both grid-rows per h-lane LD_1ROW_16B_TRANS - transpose 16 bytes/cyc to one grid-row per h-lane LD_2ROWS_8B_TRANS - transpose 8 bytes/cyc to both grid-rows per h-lane
hbuf_addr_offset	9	Additive offset from hbuf_base (not multiplied by an iterator) for calculating per-grid-row hbuf write address. **8-byte granularity**.
hbuf_stride_dim1	9	Stride multiplied by the "dimension1" selected iterator to *add* to the hbuf address calculation. **8-byte granularity**.
hbuf_stride_dim2	9	Stride multiplied by the "dimension2" selected iterator to *add* to the hbuf address calculation. **8-byte granularity**
hbuf_stride_dim3	9	Stride multiplied by the "dimension3" selected iterator to *add* to the hbuf address calculation. **8-byte granularity**
hbuf_stride_iter_id_dim1	3	Selects the iterator to multiply by hbuf_stride_dim1
hbuf_stride_iter_id_dim2	3	Selects the iterator to multiply by hbuf_stride_dim2
hbuf_stride_iter_id_dim3	3	Selects the iterator to multiply by hbuf_stride_dim3
grip_iter_id	3	selects iterator to drive **grid_row_in_pair**, i.e, to loop over the two grid rows per h-lane. Only meaningful for LD_1ROW_16B
tbuf_idx_iter_id	3	Selects the iterator to drive the **tbuf_idx** (transpose-buffer, ping-pong buffer index) to WDP. This is only meaningful for hbuf_wr_control = LD_1ROW_16B_TRANS or LD_2ROWS_8B_TRANS, i.e, when transposing the data
tbuf_col_idx_iter_id	3	Selects the iterator to drive the **tbuf_col_idx**[1:0] to WDP

		If hbuf_wr_control == LD_2ROWS_8B_TRANS, tbuf_col_idx[1:0] = selected_iter[1:0] If hbuf_wr_control == LD_1ROW_16B_TRANS, tbuf_col_idx[1:0] = {selected_iter[0], 1'b0} to load two tbuf columns otherwise, this field is not meaningful

9.1.1 H-Lane Routing

The weights-switch can move any of the 8 partitions of data (128 Bytes cycle) to any of the 8 H-lanes, and the meaningful configurations for this are discussed here.

In general, each 8-partitions of data will be distributed in one of four arrangement models, specified with the microcode field **log2_ptns_per_hlane[1:0]**:

log2 ptns per hlane	ptns / hlane	Description and usage models
0	1	**One partition routed to every H-lane**, with partition 0 routed to H-lane 0, partiton 1 to H-lane 1, etc. The unload delay is set to zero for every partition. Example use cases: • Uncompressed 1x1 weights or 3x3 weights, in which the weights database is organized such that each 128 byte AMEM word provides 8 bytes of information for each of the 16 grid-rows. • Compressed weights with a compression-block size of 4 or 8, for which the decompressed weights will be organized to provide 8 bytes of information to each of the 16 grid-rows (per 128 bytes post-decompression) • Compressed weights with a compression-block size of 16, for which decompressed weights will provide 16 bytes to each of 8 grid-rows, hence one partition to each H-lane. • Transposed MatMul with FP8 data, where each partition has 16 column-bytes which (once tranposed) correspond to 16 virtual-grid-rows. This maps to two physical grid-rows (8 VGRs/GR) on one H-lane. Hence, for transposed MatMul FP8 we send each partition to one H-Lane.
1	2	**Two (consecutive) partitions routed to each of 4 (consecutive) H-lanes** in one cycle, and then the same partitions to the other 4 (consecutive) h-lanes in the next cycle, with the pattern repeated every 2 cycles. In this model, hardware sets h-lane buffer unload_delay=0 for partition-0 of each pair, and unload_delay=1 for partition-1. Example use cases: • Transposed MatMul with FP16 data, where the first 16 byte will (once tranposed) correspond to 8 virtual grid-rows on one physical grid row, and the next 16 bytes will correspond to the 2nd grid-row mapped to the same H-Lane. So, for the application we will map two partitions to each H-Lane • Compressed weights with a compression-block size of 32, for which decompressed weights will provide 32 bytes to each of 4 grid-rows each cycle. In the first cycle, this provides 32 bytes to grid-rows 0,2,4,6 via H-lanes 0-3; in

		2nd cycle the data is sent to grid-rows 8,10,12,4 via H-lanes 4-7. Then in the 3rd cycle, data is sent to grid-rows 1,3,5,7 (H-lanes 0-3) and 4th cycle to grid-rows 9,11,13,15 (H-lanes 4-7)
2	4	**Four consecutive partitions (0-3 and 4-7) to each of 2 consecutive H-lanes**, iterating over the four sets of two H-lanes every four cycles. For this model, partitions0-3 (in each group) get unload_delay = 0-3 respectively
3	8	**All eight partitions loaded to one H-lane**, iterating over all 8 H-lanes once per 8 cycles. In this model, partition <i> is assigned unload_delay=<i> Example use cases: • "Row-shifted" MatMul, where each 128-bytes of data from AMEM belongs to a single (virtual) grid-row, hence all 8 partitions are sent to one H-lane, and iterate across the 8 H-lanes each cycle.

Table 9-1: Partititons per H-lane routing

Note: partitions may be rotated based on the data alignment from AMEM, so the partition numbers discussed in the table above are assuming no rotation, and would then be adjusted based on the AMEM data alignment by hardware. This is controlled by the microcode **field wsw_ptn_rot_en**.

9.1.2 Hbuf Write Control

This microcode field determines how the data is written into the Hbuf memories attached to each H-lane in the weights datapath. This control will be attached to every partition (16B) of data entering the Weights Switch, and the controls are routed through the switch with the associated data to the target H-lanes.

hbuf_wr_ctl	Operations
LD_1ROW_16B	Write 16 bytes/cycle to one of the two grid-rows attached to the h-lane. This will write 8 bytes to both (8-byte-wide) Hbuf banks on the targeted grid-row. The sequencer program will iterate over the two grid-rows per h-lane.
LD_2ROWS_8B	Write 8 bytes/cycle to both grid-rows attached to the h-lane. This will write 8 bytes to one Hbuf bank, on each of the two grid-rows. The sequencer program will iterate over the two Hbuf banks per grid-row.
LD_1ROW_16B_TRANS	Transpose 16 bytes/cycle This will load 16 bytes/cycle into the transpose buffer (**tbuf**), loading two (8-byte-wide) slots for one of the two grid-rows attached to the h-lane. There are two tbuf's per grid-row (ping-pong model) and each tbuf has 4 slots. The sequencer program will iterate over the tbuf (2), tbuf-slot-pairs (2) and grid-row (2) attached to each h-lane. Hardware will automatically push the data into the Hbuf once the tbuf is full (32Bytes, 4 tbuf slots)
LD_2ROWS_8B_TRANS	Transpose 8 bytes/cycle (on each of 16 grid-rows)

	This will load 8 bytes/cycle into one tbuf slot on each of two grid-rows attached to the h-lane. There are two tbuf's per grid-row (ping-pong model) and each tbuf has 4 slots. The sequencer program will iterate over the tbuf (2) and tbuf-slot index (4). Hardware will automatically push the data into the Hbuf once the tbuf is full (32Bytes, 4 tbuf slots)

Table 9-2: Hbuf write control codepoints

Not all combinations of **log2_ptns_per_hlane** and **hbuf_wr_control** are supported by hardware. The following table lists the legal combinations.

hbuf_wr_control	**Supported values of log2_ptns_per_hlane**	**Notes**
LD_1ROW_16B, LD_2ROWS_8B	0 (= 1 ptn/h-lane) 1 (= 2 ptns/h-lane) 2 (= 4 ptns/h-lane) 3 (= 8 ptns/h-lane)	All ptns_per_hlane mappings are supported for weights and "row-shifted" matmul.
LD_1ROW_16B_TRANS	1 (= 2 ptns/h-lane)	FP16 Transpose case: always loads two 16B partitions of consecutive data on each (of 4) H-lanes, since each 16B partition represents 8 values; the first partition is routed to GR_in_pair0, 2nd ptn to GR_in_pair1 attached to the H-lane Other ptns_per_hlane settings are not supported
LD_2ROWS_8B_TRANS	0 (= 1 ptn/h-lane)	FP8 Transpose case: always loads one 16B partition per H-lane, since each 16B represents 8B to each of the two grid-rows attached to one H-lane. Other ptns_per_hlane settings are not supported

Table 9-3: Supported combinations of hbuf_wr_control and log2_ptns_per_hlane

9.1.3 Grid-Row-in-Pair selection

Each H-lane feeds to Hbuf memories for two grid-rows. For the **LD_1ROW_16B** write operation, the sequencer program must specify which grid-row in each pair will receive this data. The sequencer Operations instruction includes a **grip_iter_id** field which selects one iterator to loop over the two grid-rows.

However, for the **LD_1ROW_16B_TRANS** operation, the grid-row-in-pair is generated automatically by the sequencer (rather than programmed via an iterator) in the typical case. This is because the **log2_ptns_per_hlane=1** for the typical case for FP16 transposed data, i.e., 2 partitions per H-lane. For this case, the first partition of each pair will be routed to grid-row-in-pair=0, and 2nd partition routed to grid-row-in-pair=1. This routing is done automatically by sequencer hardware, specifically for **LD_1ROW_16B_TRANS** with **log2_ptns_per_lane > 0**

The following table defines all cases for setting grid_row_in_pair. Only the supported combinations are shown.

Wr_control	log2_ptns_per_lane	grid_row_in_pair calculation
LD_1ROW_16B	any	Select using **grip_iter_idx**: grid_row_in_pair = iterator[grip_iter_idx].count[0]
LD_2ROWS_8B	any	n/a : both grid-rows written
LD_1ROW_16B_TRANS	1 (= 2 ptns/h-lane)	ptn0: grid_row_in_pair=0 ptn1: grid_row_in_pair=1
LD_2ROWS_8B_TRANS	0 (= 1 ptn/h-lane)	n/a : both grid-rows written

Table 9-4: Grid-row-in-pair calculation for HBUF write

9.1.4 Hbuf Write Address calculation

For writing to the per-grid-row Horizontal Data Buffers (Hbuf), the addresses have **8-byte granularity**, which may be expressed as the concatenation of **hbuf_address_in_bank** (8 bits) and **hbuf_bank** (1 bit). The byte offset is not used for writes:

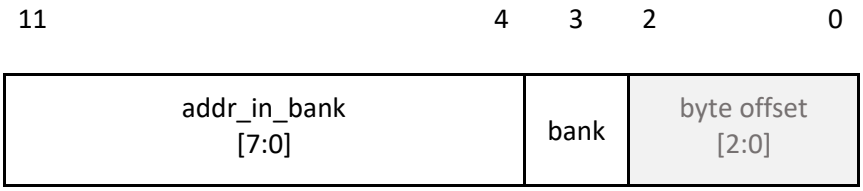


Table 9-5: HBuf write-address format

Hbuf write-address calculation sums the following components

- **hbuf_base** address is used for “chunky” FIFO operation, maintained by hardware. This is reset to zero, and thereafter counts forward in units of hbuf_block_size bytes (wrapping around the Hbuf memory space automatically). The hbuf_block_size is programmed independently in every trip microprogram. For example, in one trip, the hbuf_block_size might be 64 bytes (per grid-row), and the next trip might be 1024 bytes (per grid-row). The hbuf_base has 16-byte granularity, spanning one 8-byte word in both banks.
- Up to three **stride*iterator** offsets provided in the sequencer microinstruction. This allows up to three-dimensional decomposition of the input channels within each hbuf_block.
- An optional **hbuf_offset** provided in each sequencer microinstruction. This supports loop-unrolling to calculate an effective offset without a stride*iterator

The computed offsets are added to the hbuf_base, modulo the hbuf memory size (per grid-row), to form the write-address to the hbuf memory in each grid-row.

```
#compute effective hbuf address, with 8-byte granularity
#First, select per-dimension iterator values
dim1_iter_value = sequencer_iter[op_instr.dim1_iter_id].value[7:0]  # uses 8 bits
dim2_iter_value = sequencer_iter[op_instr.dim2_iter_id].value[7:0]
dim3_iter_value = sequencer_iter[op_instr.dim3_iter_id].value[7:0]

# compute iterator * stride for each dimension.
# In case of overflow, only the 9 LSBits are kept for each product (and 8B granular)
dim1_prod[8:0] = op_instr.hbuf_stride_dim1[8:0] * dim1_iter_value[8:0]
dim2_prod[8:0] = op_instr.hbuf_stride_dim2[8:0] * dim2_iter_value[8:0]
dim3_prod[8:0] = op_instr.hbuf_stride_dim3[8:0] * dim3_iter_value[8:0]

#compute sum of hbuf_base, per-instruction fixed offset, and per-dimension offsets
hbuf_addr[11:3] =                   #  8-byte granular addresses
    (hbuf_base[11:4, 1'b0] +        #  hbuf_base address maintained by hardware
    op_instr.hbuf_offset[8:0] +     #  hbuf_offset field is also 8-byte granular
    dim1_prod[8:0] +
    dim2_prod[8:0] +
    dim3_prod[8:0]) % 512  # modulo hbuf_size (8B granular), so address can wrap

# Split hbuf_addr into starting addr_in_bank and bank fields
# When writing in transposed mode, the 2 lsbits of addr_in_bank are forced to zero

hbuf_addr_in_bank = hbuf_addr[11:4]  # byte-granular addressing nomenclature
if (wr_control == LD_2ROWS_8B_TRANS or wr_control == LD_1ROW_16B_TRANS)
    hbuf_addr_in_bank &= ~0x3
hbuf_bank         = hbuf_addr[3]
```

NOTE: when writing in transposed mode, the two LSBits of **hbuf_addr_in_bank** are forced to zero by hardware, since output of the Transpose Buffer (TBUF) is written to four Hbuf words on 4-Hbuf-word boundaries.

NOTE: When **log2_ptns_per_grid_row** > 0, multiple partitions-worth of data will be written to the same grid-row's Hbuf memories, starting from a single word of AMEM data. For this case, the sequencer will automatically increment the address for each successive partitition mapped to the same H-lane:

- for **LD_1ROW_16B**: each partition's data is written to both banks in a particular grid-row. Successive partitions (mapped to the same h-lane) will increment **hbuf_addr_in_bank** automatically by the sequencer

  For example, when all 8 partitions are targeted to the same H-lane and same grid-row using **LD_1ROW_16B**, partition 0 data will be assigned the (computed) address_in_bank = A, partition 1 will get A+1, …,. partition 7 will get address_in_bank = A+7.

- for **LD_2ROWS_8B**: each partition's data is written to one bank in both grid-rows attached to an h-lane. Considering the concatenation {**hbuf_addr_in_bank**, **hbuf_bank**} as the logical address of an 8-byte write to an Hbuf, then successive partitions mapped to the same H-lane will have this logical address incremented automatically by the sequencer.

For example, when all 8 partitions are targeted to the same H-lane using **LD_2ROWS_8B**, partition 0 will be assigned the computed starting {hbuf_address_in_bank=A, hbuf_bank=B}, partition 1 will get {A,B}+1, …, partition 7 will get {A,B}+7.

The write address adjustments done by hardware, for multiple partitions mapped per hlane, are sumarized in the following table. The curly braces denote bitwise concatenation (verilog-style notation):

Wr_control	log2_ptns_per_lane	hbuf_write address adjustment for ptns in hlane
LD_1ROW_16B	0 (= 1 ptn/h-lane)	addr_in_bank: no change bank = don't care (set to zero)
	1 (= 2 ptns/h-lane)	ptn0_addr_in_bank = hbuf_addr_in_bank ptn1_addr_in_bank = hbuf_addr_in_bank+1 bank = don't care (set to zero)
	2 (= 4 ptns/h-lane)	ptn0_addr_in_bank = hbuf_addr_in_bank ptn1_addr_in_bank = hbuf_addr_in_bank+1 ptn2_addr_in_bank = hbuf_addr_in_bank+2 ptn3_addr_in_bank = hbuf_addr_in_bank+3 bank = don't care (set to zero)
	3 (= 8 ptns/h-lane)	ptn0_addr_in_bank = hbuf_addr_in_bank ptn1_addr_in_bank = hbuf_addr_in_bank+1 ptn2_addr_in_bank = hbuf_addr_in_bank+2 ptn3_addr_in_bank = hbuf_addr_in_bank+3 … ptn7_addr_in_bank = hbuf_addr_in_bank+7 bank = don't care (set to zero)
LD_2ROWS_8B	0 (= 1 ptn/h-lane)	addr_in_bank = hbuf_addr_in_bank bank = hbuf_bank
	1 (= 2 ptns/h-lane)	ptn0_addr_in_bank = hbuf_addr_in_bank bank = hbuf_bank ptn1_{addr_in_bank, bank} = {hbuf_addr_in_bank, hbuf_bank} + 1
	2 (= 4 ptns/h-lane)	ptn0_addr_in_bank = hbuf_addr_in_bank bank = hbuf_bank ptn1_{addr_in_bank, bank} = {hbuf_addr_in_bank, hbuf_bank} + 1 ptn2: {addr_in_bank, bank} = {hbuf_addr_in_bank, hbuf_bank} + 2 ptn3: {addr_in_bank, bank} = {hbuf_addr_in_bank, hbuf_bank} + 3
	3 (= 8 ptns/h-lane)	ptn0_addr_in_bank = hbuf_addr_in_bank bank = hbuf_bank ptn1_{addr_in_bank, bank} = {hbuf_addr_in_bank, hbuf_bank} + 1 ptn2_{addr_in_bank, bank} = {hbuf_addr_in_bank, hbuf_bank} + 2 … ptn7_{addr_in_bank, bank} = {hbuf_addr_in_bank, hbuf_bank} + 7
LD_1ROW_16B_TRANS	1 (= 2 ptns/h-lane)	addr_in_bank[1:0] = 0; # 2 LSBits forced to zero ptn0 written to grid_row_in_pair 0, ptn1 to grid-row 1)
LD_2ROWS_8B_TRANS	0 (= 1 ptn/h-lane)	addr_in_bank[1:0] = 0; # 2 LSBits forced to zero

Table 9-6: hbuf_address adjments for multiple partitions mapped to the same H-lane

9.1.5 Hbuf FIFO Write Operation

The Hbuf memories operate as FIFOs, where the number of entries is incremented and decremented in chunks of **hbuf_block_size**. The current remaining space available in the Hbuf is maintained in the hbuf_wr_credits register by hardware. (The credits are intialized to the full hbuf size by hardware.)

At the start of writing each block, the hbuf_wr_credits are checked and decremented by the hbuf_block_size (under program control). When an entire "block" of data has been written to hbuf memories, the hbuf_base_address is incremented by the hbuf_block_size, and the number of bytes written is sent to the Grid Horizontal Data Read sequencer (as hbuf_rd_credits) to advertise the availablity of the block of data. The hbuf_base address wraps around the hbuf memory size automatically.

For example, an hbuf_block might be configured for 1024 bytes (per grid-row). The WDP Sequencer will check that it has 1024 credits at the start of a block, and decrement that amount. Once it has written all 1024 bytest to every grid-row, the sequencer will signal the available read-credits to GH sequencer, and advance the h_buf_base by 1024. Later, After consuming that data, GH Read sequencer signals a decrement (by the block size, i.e, 1024) back to WDP Sequencer.

Block size programming:

- For row-shifted MatMul, hbuf_block_size is typically 1024 bytes (128 bytes * 8 VirtualGridRows).
- For transposed MatMul, the hbuf_block_size is typically 64 bytes (8 bytes * 8 VirtualGridRows).
- For 1x1/Linear, or 3x3 conv, the hbuf_block_size depends on the weights database organization in memory, with a typical minimum of 64 bytes.

```
# maintain hbuf_base and credits
if op_instr.hbuf_block_start and op_instr.hbuf_block_iter_mask != 0:
    block_start = True  # init
    for i in range(num_iters):
        if (op_instr.hbuf_block_iter_mask[i]) and iter[i].value != 0:
            block_start = False
    if block_start:
        stall if hbuf_write_creds < hbuf_block_size
        decrement hbuf_write_creds by hbuf_block_size


if op_instr.hbuf_block_end and op_instr.hbuf_block_iter_mask != 0:
    block_done = true  # init
    for i in range(num_iters):
        if (op_instr.hbuf_block_iter_mask[i]) and !iters_eq_nloops[i]:
            block_done = False

    if block_done:
        hbuf_base += op_instr.hbuf_block_size
        increment GH sequencer hbuf_read_credits by hbuf_block_size
```

9.1.6 Zero-mask selection

The WDP Sequencer op instruction provides two zero-mask index selections per partition, and common zero-mask-enable, as well as a configuration-update enable. If the configuration update is enabled, the zero-mask selections for the microinstruction are applied to the WDP, otherwise WDP uses the previously configured zero mask settings.

If the zero-mask is disabled, no zero masking is applied. Otherwise, the selected, per-partition zero mask is applied.

Zero-masks are associated with H-lanes. So, the sequencer programming must assign the per-partition zero-mask index so that it picks up the correct mask for the assigned H-lane. For example, if partition 0 is routed to H-lane 0 in one cycle, and routed to H-lane 4 in another cycle, then the zero-mask index (for partition 0 data) will be applicable to H-lane 0 in the first cycle, and to H-lane 4 in the second cycle.

In most cases, a single zero-mask index is sufficient, consistent with a static zero-mask for the duration of processing a tile. However, in the **LD_1ROW_16B_TRANS** case, which is used for FP16 transposed data, two zero mask indices are needed for each partition. This is because two partitions will share a single H-lane, and that one H-lane is then routed to two different grid-rows on alternating cycles. So in effect, we need to multiplex the grid-row selection onto the zero-mask index. To support this, the op instruction provides two zero-mask index values (A & B). During even cycles, the data is routed to grid-row 0 (in the pair), and would use zero-mask index A, while during odd cycles, data is routed to grid-row 1, and would use zero-mask index B.

This special mask selection is performed automatically by the sequencer hardware for the LD_1ROW_16B_TRANS write control operation.

9.2 WDP ALGORITHM FOR 1X1 WEIGHTS LOADING

9.2.1 WDP 1x1 weights loading with fine-grained channel packing

Fine-grained channel packing is the most memory-efficient and lowest latency method for arranging weights, in which each 128-byte AMEM word contains 8 bytes of weights for 16 different grid-rows. Each HBUF block contains weights for 8 output channels (per grid-row), so it takes 8 cycles to write an HBUF block, after which every grid-row will have received 8 bytes for each of 8 output channels (which are equivalent to virtual grid-rows).

Hbuf block size = 8*8=64B

Each cycle, write 16B to every H-lane, which is then relayed as 8 bytes to each of the 16 grid-rows.

This applies for either LnsI4F3 or LnsI5F10, when packed with 8 bytes (of input channel weights) per output channel

Hbuf write operation: **LD_2ROWS_8B**

H= Horizontal Lane, D=Delay in weights switch buffer, Gr=target grid-rows in h-lane

Cout	HBUF Block	HBUF addr in bank	HBUF bank	Hlane, Delay, Grid-row(s) per partition							
				Ptn 7	Ptn 6	Ptn 5	Ptn 4	Ptn 3	Ptn 2	Ptn 1	Ptn 0
0	0	0	0	H 7 D 0 Gr 1,0	H 6 D 0 Gr 1,0	H 5 D 0 Gr 1,0	H 4 D 0 Gr 1,0	H 3 D 0 Gr 1,0	H 2 D 0 Gr 1,0	H 1 D 0 Gr 1,0	H 0 D 0 Gr 1,0
1			1								
2		1	0								
3			1								
4		2	0								
5			1								
6		3	0								
7			1								

```
# WDP sequencer alg for fine-grained 1x1 Weights (8bit/16bit weights):
# Hbuf block size = 64B (per grid-row)

for hblock_iter in range (hbuf_blocks):
    for cout_iter in range (8):        # 8 output channels (per grid-row)
        write_hbuf(
            is_16bit = is_16bit_weights,
            hbuf_write_ctl = LD_2ROWS_8B,
            hbuf_block_size = 4,               # *16 = 64bytes
            hbuf_block_start_en = 1,
            hbuf_block_end_en = 1,
            hbuf_block_iter_mask = MASK(cout_iter)

            hbuf_stride_dim1 = 1,               # *8 = 8bytes stride per cout
            hbuf_stride_dim1_iter_id = cout_iter,
            log2_ptns_per_hlane = 0)            # one partiton to each hlane
```

Figure 9-2: WDP Seq pseudo-code for fine-grained 1x1 weights loading

9.3 WDP ALGORITHMS FOR MATMUL (DATA X DATA) ROW-SHIFTED.

FP8 MatMul Row-shifted transfer pattern for one HBUF block (128 cycles). Each cycle, write 128B into one H-lane, which is shifted over 8 cycles into one grid-row; repeat for each of 16 physical grid-rows, and then loop over 8 virtual grid-rows per physical grid-row

Hbuf block size = 128 * 8 = 1024B

Hbuf write operation: **LD_1ROW_16B**

Loop 0: VGR in GR	Loop 1: GR in Hlane	Loop 2: Hlane	Grid Row	Virtual Grid Row	HBUF addr in bank	HBUF bank	Hlane, Delay, Grid-row(s) , addr_in_bank per partition			
							Hlane	Delay	Grid Row in Pair	Addr in Bank
0	0	0	0	0	0-7	0 & 1	0	D=ptn	0	Addr=ptn
		1	2	16			1			
		2	4	32			2			
		…	…	…			…			
		7	14	112			7			
	1	0	1	8	0-7	0 & 1	0	D=ptn	1	Addr=ptn
		1	3	24			1			
		2	5	40			2			
		…	…	…			…			
		7	15	120			7			
1	0	0	0	1	8-15	0 & 1	0	D=ptn	0	Addr = 8+ptn
		1	2	17			1			
		2	4	33			2			
		…	…	…			…			
		7	14	113			7			
	1	0	1	9	8-15	0 & 1	0	D=ptn	1	Addr = 8+ptn
		1	3	25			1			
		2	5	41			2			
		…	…	…			…			
		7	15	121			7			
			…	…	…	…				
7	0	0	0	7	56-63	0 & 1	0	D=ptn	0	Addr = 56+ptn
		1	2	23			1			
		…	…	…						
		7	14	119			7			
	1	0	1	15	56-63	0 & 1	0	D=ptn	1	Addr = 56+ptn
		1	3	31			1			
		…	…	…			…			
		7	15	127			7			

```
FP8 / FP16 MatMul Row-shifted (basic algorithm)
Hbuf write Operation: LD_1ROW_16B

# hbuf_block is block of bytes per gridrow which need to be buffered.
# This is the chunky-flow-control unit for managing hbuf memory
hbuf_block_size = 8 * 128  # bytes

for hbuf_block in hbuf_blocks:          # i0: hbuf_block index
    hbuf_wr_credits_decr(hbuf_block_size)  # deduct credits used for entire block
    for vgr in range (8):               # i1: vgr offset in each grid-row
        for grip in range(2):           # i2: grid-row-in-pair in hlane
            for hlane in range(8):      # i3: iterate over hlanes
                mdata.lin2log_en,eb_adj_val, zero_mask_en, zero_mask_idx = in_args.*
                mdata.fp16 = in_args.fp16   # only difference for fp8/fp16 MM shifted!

                # calc hlane index & delay for each partition.
                # the following does not use iterators; performed by decode logic
                for ptn in range(8):
                    mdata.hidx_ptn[ptn] = hlane
                    mdata.delay_ptn[ptn] = (ptn + amem_ptn_rot)%8
                    mdata.wr_control[ptn] = LD_1ROW_16B
                    mdata.gridrow_in_pair[ptn] = grip
                    mdata.hbuf_addr_in_bank[ptn] = hbuf_base + vgr*8 + ptn

    # Inform grid-control sequencers when a full hbuf_block has been written
    hbuf_rd_credits_incr(hbuf_block_size)
    hbuf_base += hbuf_block_size/16 # units of 16 == granularity of addr_in_bank
```

9.4 WDP ALGORITHMS FOR TRANSPOSED MATMUL

9.4.1 FP8 Transpose

```
# WDP Sequencer algorithm for FP8 Transposed
# Hbuf block size = 64B (per grid-row) which corresponds to 8 AMEM data words (1024B)

for hblock_iter in range (hbuf_blocks):
    for tbuf_idx_iter in range (2):                    # tbufs
        for tbuf_col_idx_iter in range (4):            # columns in tbuf
            write_hbuf(hbuf_write_ctl=LD_2ROWS_8B_TRANS,
                       log2_ptns_per_hlane = 0,
                       tbuf_idx_iter_id = tbuf_idx_iter,
                       tbuf_col_idx_iter_id = tbuf_col_idx_iter,
                       hbuf_stride_dim1 = 1,           # 1 bank stride
                       hbuf_stride_dim1_iter_id = tbuf_idx_iter,

                       hbuf_block_size = 4,            # *16 = 64bytes
                       hbuf_block_start_en = 1,
                       hbuf_block_end_en = 1,
                       hbuf_block_iter_mask = MASK(tbuf_idx_iter, tbuf_col_idx_iter)
```

Figure 9-3: WDP Seq pseudo-code for FP8 Transposed Matmul

FP8 Transposed data is written from AMEM to Tbufs to Hbufs with the following summary arrangement for each Hbuf block of 64 Bytes per grid-row. The FP8 data arrives (to each grid-row) for 8 Rows (A…H) with 8 columns (0..7) per row, and is transposed to 8 rows (0..7) with 8 columns (A..H) per row

TBUF										**HBUF**									
tbuf index	tbuf col idx	byte offset								byte offset								hbuf bank	hbuf word in bank
		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
0	0	A7	A6	A5	A4	A3	A2	A1	A0	D1	C1	B1	A1	D0	C0	B0	A0	0	0
	1	B7	B6	B5	B4	B3	B2	B1	B0	D3	C3	B3	A3	D2	C2	B2	A2		1
	2	C7	C6	C5	C4	C3	C2	C1	C0	D5	C5	B5	A5	D4	C4	B4	A4		2
	3	D7	D6	D5	D4	D3	D2	D1	D0	D7	C7	B7	A7	D6	C6	B6	A6		3
1	0	E7	E6	E5	E4	E3	E2	E1	E0	H1	G1	F1	E1	H0	G0	F0	E0	1	0
	1	F7	F6	F5	F4	F3	F2	F1	F0	H3	G3	F3	E3	H2	G2	F2	E2		1
	2	G7	G6	G5	G4	G3	G2	G1	G0	H5	G5	F5	E5	H4	G4	F4	E4		2
	3	H7	H6	H5	H4	H3	H2	H1	H0	H7	G7	F7	E7	H6	G6	F6	E6		3

Table 9-7: FP8 Transpose Data Mapping to TBUF and HBUF, per-grid-row

9.4.1.1 FP8 Transpose – WDP controls for one Hbuf block (8 cycles)

Below is the FP8 Transpose pattern for one HBUF block (8 cycles). All addresses are offset from **hbuf_base**. Hlanes may be rotated by received **ptn_rot[2:0]** if **wsw_ptn_rot_en** is set

HBUF write operation: **LD_2ROWS_8B_TRANS**

Tensor Row	HBUF addr in bank	HBUF bank	Tbuf idx	Tbuf col idx	Hlane, Delay, Grid-row(s) per partition							
					Ptn 7	Ptn 6	Ptn 5	Ptn 4	Ptn 3	Ptn 2	Ptn 1	Ptn 0
A	0 (will become 0..3 when tbuf is automatically written to hbuf)	0	0	0	H 7 D 0 Gr 1,0	H 6 D 0 Gr 1,0	H 5 D 0 Gr 1,0	H 4 D 0 Gr 1,0	H 3 D 0 Gr 1,0	H 2 D 0 Gr 1,0	H 1 D 0 Gr 1,0	H 0 D 0 Gr 1,0
B				1								
C				2								
D				3								
E		1	1	0								
F				1								
G				2								
H				3								

Table 9-8: LD_2ROWS_8B_TRANS sequence

9.4.2 FP16 Transpose

```
# WDP Sequencer algorithm for FP16 Transposed
# Hbuf block size = 64B (per grid-row) which corresponds to 8 AMEM data words (1024B)

for hblock_iter in range (hbuf_blocks):
    for tbuf_idx_iter in range (2):                     # tbufs
        for tbuf_col_idx_iter in range (2):             # column-pairs in tbuf
            for hlane_group_iter in range (2):          # two hlane groups (0-3,4-7)
                write_hbuf(hbuf_write_ctl=LD_1ROW_16B_TRANS,
                        log2_ptns_per_hlane = 1,        # 2 partitions per hlane
                        hlane_iter_id = hlane_group_iter,
                        tbuf_idx_iter_id = tbuf_idx_iter,
                        tbuf_col_idx_iter_id = tbuf_col_idx_iter,
                        hbuf_stride_dim1 = 1,           # 1 bank stride,
                        hbuf_stride_dim1_iter_id = tbuf_idx_iter,

                        hbuf_block_size = 4,            # *16 = 64bytes
                        hbuf_block_start_en = 1,
                        hbuf_block_end_en = 1,
                        hbuf_block_iter_mask = MASK(tbuf_idx_iter, tbuf_col_idx_iter,
                                                    hlane_group_iter))
```

Figure 9-4: WDP Seq pseudo-code for FP16 Transposed Matmul

FP8 Transposed data is written from AMEM to Tbufs to Hbufs with the following summary arrangement for one block. The FP16 data arrives (to each grid-row) for 4 data rows (A…D) with 8 columns (0..7) per row, and is transposed to 8 data-rows (0..7) with 4 columns (A..D) per row. Hlanes may be rotated by received **ptn_rot[2:0]** if **wsw_ptn_rot_en** is set.

TBUF						HBUF					
tbuf index	tbuf col idx	tbuf byte offset				hbuf byte offset				hbuf bank	hbuf word in bank
		7,6	5,4	3,2	1,0	7,6	5,4	3,2	1,0		
0	0	A3	A2	A1	A0	B4	A4	B0	A0	0	0
	1	A7	A6	A5	A4	B5	A5	B1	A1		1
	2	B3	B2	B1	B0	B6	A6	B2	A2		2
	3	B7	B6	B5	B4	B7	A7	B3	A3		3
1	0	C3	C2	C1	C0	D4	C4	D0	C0	1	0
	1	C7	C6	C5	C4	D5	C5	D1	C1		1
	2	D3	D2	D1	D0	D6	C6	D2	C2		2
	3	D7	D6	D5	D4	D7	C7	D3	C3		3

Table 9-9: FP16 Transpose Data Mapping to TBUF and HBUF

9.4.2.1 FP16 Transpose – WDP controls for one HBUF block (8 cycles)

HBUF write operation: **LD_1ROW_16B_TRANS**

Tensor Row, logical cols	HBUF addr in bank	HBUF bank	Tbuf idx	Tbuf col idx	H=Hlane, D=Delay, Gr=Grid-row-in-pair per partition							
					Ptn 7	Ptn 6	Ptn 5	Ptn 4	Ptn 3	Ptn 2	Ptn 1	Ptn 0
A 63:0	0 (will become 0..3 when tbuf is automatically written to hbuf)	0	0	0	H 3 D1 Gr 1	H 3 D0 Gr0	H 2 D1 Gr 1	H 2 D0 Gr 0	H1 D1 Gr 1	H 1 D0 Gr 0	H 0 D1 Gr 1	H 0 D0 Gr0
A 127:64					H 7 D1 Gr 1	H 7 D0 Gr 0	H6 D1 Gr 1	H 6 D0 Gr 0	H5 D1 Gr 1	H5 D0 Gr 0	H4 D1 Gr 1	H 4 D0 Gr 0
B 63:0				2	H 3 D1 Gr 1	H 3 D0 Gr0	H 2 D1 Gr 1	H 2 D0 Gr 0	H1 D1 Gr 1	H 1 D0 Gr 0	H 0 D1 Gr 1	H 0 D0 Gr0
B 127:64					H 7 D1 Gr 1	H 7 D0 Gr 0	H6 D1 Gr 1	H 6 D0 Gr 0	H5 D1 Gr 1	H5 D0 Gr 0	H4 D1 Gr 1	H 4 D0 Gr 0
C 63:0		1	1	0	H 3 D1 Gr 1	H 3 D0 Gr0	H 2 D1 Gr 1	H 2 D0 Gr 0	H1 D1 Gr 1	H 1 D0 Gr 0	H 0 D1 Gr 1	H 0 D0 Gr0
C 127:64					H 7 D1 Gr 1	H 7 D0 Gr 0	H6 D1 Gr 1	H 6 D0 Gr 0	H5 D1 Gr 1	H5 D0 Gr 0	H4 D1 Gr 1	H 4 D0 Gr 0
D 63:0				2	H 3 D1 Gr 1	H 3 D0 Gr0	H 2 D1 Gr 1	H 2 D0 Gr 0	H1 D1 Gr 1	H 1 D0 Gr 0	H 0 D1 Gr 1	H 0 D0 Gr0
D 127:64					H 7 D1 Gr 1	H 7 D0 Gr 0	H6 D1 Gr 1	H 6 D0 Gr 0	H5 D1 Gr 1	H5 D0 Gr 0	H4 D1 Gr 1	H 4 D0 Gr 0

Table 9-10: LD_1ROW_16B_TRANS sequence

9.5 WDP ALGORITHM FOR 3X3 CONV

TBD. Similar to writing 1x1 weights - we can write 8B to every grid row each cycle, and repeat for a block size. Key difference from 1x1 is that Hbuf block size may be rounded up to a convenient multiple of 16 bytes (per grid-row)

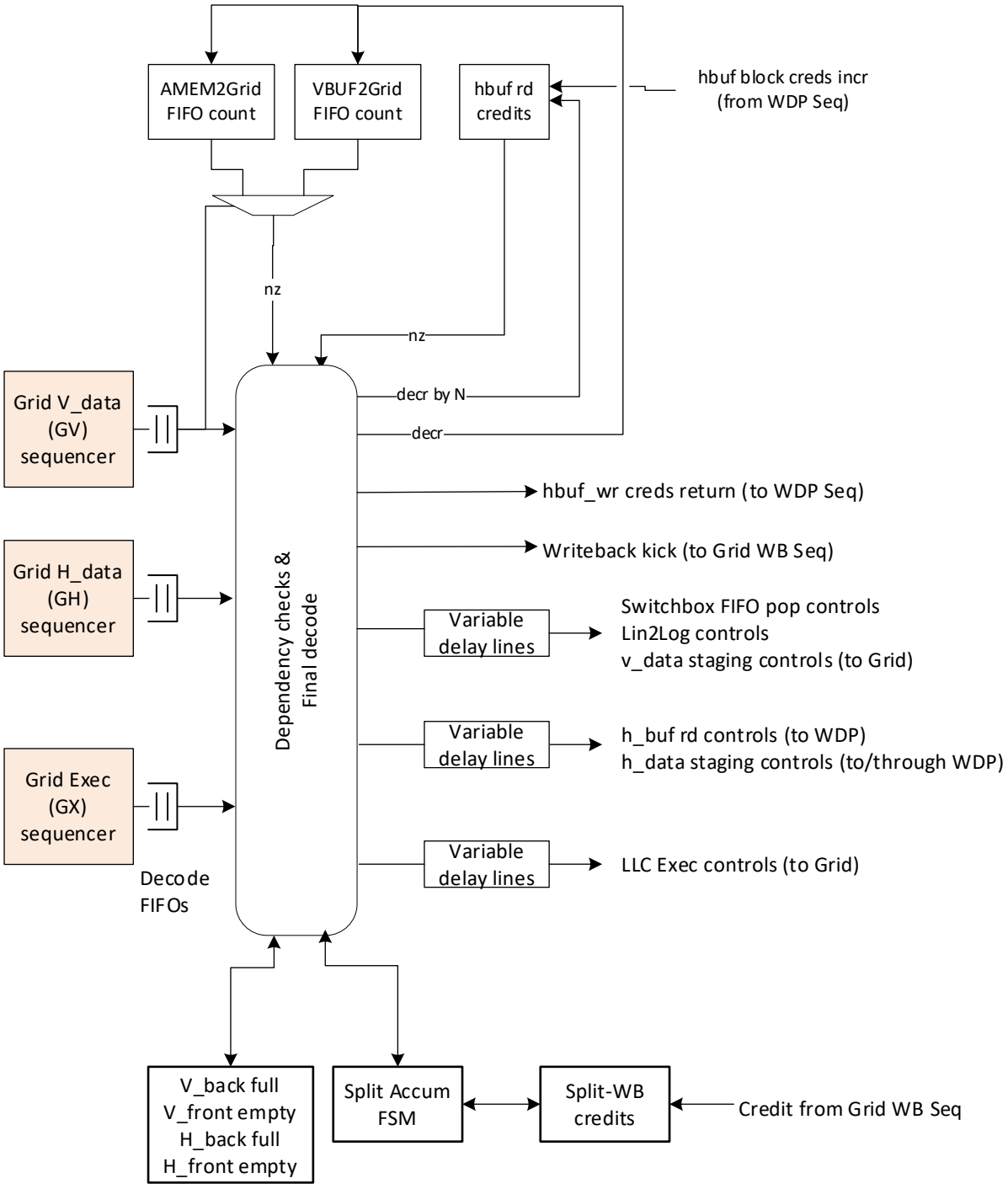
9.6 WEIGHTS-SWITCH PROGRAMMING NOTES (BACKGROUND INFO)

h-lane[ptn] and delay[ptn] are a function of partitions_per_hlane, the specific partition, and the hlane iterator.

Simplified (limited) H-lane programming

Log2 Ptns Per hlane	#ptns / Hlane	Target H-lane for each of Ptn 7..0 (Shown as sequences vertically)								H-lane cfg h[ptn] =	H-lane delay for each Ptn 7..0								Dly[ptn]
		Ptn 7	Ptn 6	Ptn 5	Ptn 4	ptn 3	ptn 2	Ptn 1	Ptn 0		ptn 7	ptn 6	ptn 5	ptn 4	ptn 3	ptn 2	ptn 1	ptn 0	
0	1	7	6	5	4	3	2	1	0	h[ptn] = ptn[2:0]	0	0	0	0	0	0	0	0	dly[ptn]=0
1	2	3	3	2	2	1	1	0	0	h[ptn] ={iter[0], ptn[2:1]} (fp16_transp case)	1	0	1	0	1	0	1	0	dly[ptn]= ptn[0]
		7	7	6	6	5	5	4	4										
2	4	1	1	1	1	0	0	0	0	h[ptn] ={iter[1:0], ptn[2]}	3	2	1	0	3	2	1	0	dly[ptn]= ptn[1:0]
		3	3	3	3	2	2	2	2										
		5	5	5	5	4	4	4	4										
		7	7	7	7	6	6	6	6										
3	8	0	0	0	0	0	0	0	0	h[ptn] = iter[2:0]	7	6	5	4	3	2	1	0	dly[ptn]= ptn[2:0]
		1	1	1	1	1	1	1	1										
		2	2	2	2	2	2	2	2										
		3	3	3	3	3	3	3	3										
		4	4	4	4	4	4	4	4										
		5	5	5	5	5	5	5	5										
		6	6	6	6	6	6	6	6										
		7	7	7	7	7	7	7	7										

10 TOP-OF-GRID CONTROL SEQUENCERS



There are three co-ordinated sequencers for operating the LLC Grid, conceptually located at the “top” of the grid:

- Grid-Horizontal-Data: (GH) coordinates movement of horizontal data from the weights buffer memories to the weights-staging buffers in the LLCs.
- Grid-Vertical-Data (GV) coordinates movement of vertical data from the Switchbox to data-staging buffers in the LLCs.
- Grid-Execution (GX) and controls the execution of the LLCs to compute dot products and write data into the accumulators

Each of these sequencers runs an independent microprogram to produce controls buffered in per-sequencer FIFOs. Hardware then coordinates the timing of the generated controls from all three sequencers to properly converge on the LLC Grid.

For example, in MatMul, the execution depends on having data ready in the V_staging buffer, and having data available in the Hbuf memories (in WDP), so the Execution controls will be implicitly dependent on those conditions before the operation is validated.

10.1 GH (GRID HORIZONTAL DATA) SEQUENCER

GH Sequencer Op instruction format:

Field	Width	Description
opcd	1	0: Nop 1: HBUF_RD_VLD -- issue hbuf data read according to **hbuf_rd_cmd**
hbuf_rd_cmd	3	**RD_3X3**: read 9 x 1-byte values across two Hbuf banks for 3x3 kernels. Used only for 3x3 convolutions **RD_1X1_MATMUL_FP8**: read 8 x 1-byte values for 1x1 conv, linear layers or MatMul data (FP8) **RD_1x1_MATMUL_FP16**: read 4 x 2-byte values for 1x1 conv, linear layers or MatMul data (FP16) **RD_TRANS_1X1_MATMUL_FP8**: read 8 x 1-byte values as 4B from each of two Hbuf banks for **transposed** 1x1 or linear layer weights or MatMul data (FP8) **RD_TRANS_1X1_MATMUL_FP16**: read 4 x 2-byte values as 4B from each of two Hbuf banks for **transposed** 1x1 or linear layer weights or MatMul data (FP16)
end_grid_row_idx	4	Supports a subset of active grid-rows each cycle. The hbuf memories are not read for inactive rows, and the contents of h_staging buffers will be undefined.
log2_filters_per_row	2	1, 2, 4 or 8 filters per grid-row; applicable for 3x3 mode only. Supports narrow tensors where multiple filters (output channels) may be mapped

		to the same grid row. Needed by hardware to direct each 3x3 kernel to the staging “back” buffer for just a subset of partitions. Only meaningful for 3x3 mode.
h_staging_filter_iter_id	3	Selects the iterator to loop over the filters per grid-row, to direct each 3x3 kernel to the staging (back) buffers for the partitions associated with that filter (output channel). Note: This same iterator would typically be used for calculating the hbuf address of a particular 3x3 kernel. Only meaningful for 3x3 mode. This field must be set even for wider tensors with just one filter per grid row (in which case the associated iterator only counts to 1).
h_staging_done_en	1	When set, and the iterator indicated by h_staging_filter_iter_id has its ending loop value, this indicates that the h_staging “back” buffer is full, which allows data to be moved from the back to front staging buffer in the llc_grid. Only meaningful for 3x3 mode.
hbuf address calculation fields		
hbuf_block_size	9	Number of bytes in hbuf_block (per grid-row), in **units of 16-bytes.** 16 bytes corresponds to one entry in both banks of hbuf memory. This is the block size used for “chunky” flow control with the weights datapath (which writes into hbuf logical FIFOs). 9 bits allows values up to the entire size of hbuf Note: LnsI5F3 data is counted as one “byte”, LnsI5F10 is two “bytes”
hbuf_block_start_en	1	Instructs the sequencer to require available data in the hbuf (logical) FIFOs, when all iterators indicated by hbuf_block_iter_mask are zero. By convention, these FIFOs are incremented with block-size “chunky” credits, so it is only necessary to look for nonzero credits at the start of a block to proceed with all reads for the entire block
hbuf_block_end_en	1	Instructs the sequencer that it has fully consumed a block of hbuf data, when all iterators indicated by hbuf_block_iter_mask have reached their ending loop value. When this occurs, the hbuf_rd_credits is decremented by hbuf_block_size, (returning credits also to WDP) and the hbuf_base read address is advanced by the hbuf_block_size
hbuf_block_iter_mask	6	Mask of iterator(s) which control start/ending an hbuf block. Instructs the sequencer which iterators should be zero for the hbuf_block_start function, and/or which iterators should have their ending loop value for the hbuf_block_end function When the mask itself is zero, no iterators are examined, and hbuf_block_start/end is completely controlled by the start/end bits
hbuf_addr_offset	12	Additive offset from hbuf_base (not multiplied by an iterator) for calculating per-grid-row hbuf read address. Byte-address granularity. Useful for loop-unrolling. 11b supports unrolling within a 2048-byte block

hbuf_stride_dim1	12	Stride multiplied by the “dimension1” selected iterator to *add* to the hbuf address calculation. Byte granularity
hbuf_stride_dim2	12	Stride multiplied by the “dimension2” selected iterator to *add* to the hbuf address calculation. Byte granularity
hbuf_stride_dim3	12	Stride multiplied by the “dimension3” selected iterator to *add* to the hbuf address calculation. Byte granularity
hbuf_stride_iter_id_dim1	3	Selects the iterator to multiply by hbuf_stride_dim1
hbuf_stride_iter_id_dim2	3	Selects the iterator to multiply by hbuf_stride_dim2
hbuf_stride_iter_id_dim3	3	Selects the iterator to multiply by hbuf_stride_dim3

10.1.1 Hbuf Read Address calculation

For reading from the per-grid-row Horizontal Data Buffers (Hbuf), the addresses have **byte granularity**, which may be expressed as the concatenation of **hbuf_address_in_bank** (8 bits), **hbuf_bank** (1 bit), and **byte offset** (3 bits). The byte offset is used for indexing 3x3 kernels which are 9 logical bytes. For 1x1 weights and MatMul data, the byte offset is set to zero.

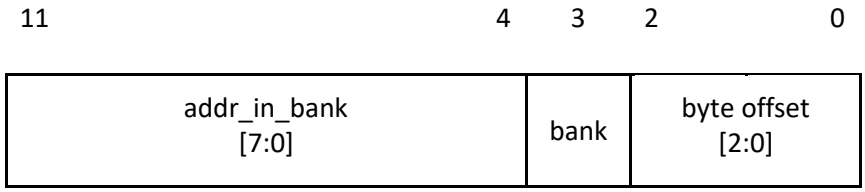


Table 10-1: HBuf read-address format

HBuf read-address is calculated as the sum of the following components:

- hbuf_base address used for "chunky" FIFO operation, maintained by hardware. This is reset to zero, and thereafter counts forward in units of **hbuf_block_size** bytes.
- up to three stride*iterator offsets from the h_data sequencer microinstruction.
- an optional per-instruction hbuf_offset from the h_data sequencer microinstruction. This supports loop-unrolling to calculate an effective offset without a stride*iterator

Note: all hbuf addresses computed here are for "logical" bytes, where each logical byte is an I5F3 value (which is 9 bits) or half of an I5F10 value (which is 16 bits).

The computed offsets are added to the hbuf_base, modulo the hbuf memory size (per grid-row), to form the (byte-granular) address to the hbuf memory in each grid-row.

```
#compute effective hbuf address, with byte-granularity
#First, select per-dimension iterator values
dim1_iter_value = sequencer_iter[op_instr.dim1_iter_id].count[7:0]  # uses 8 bits
dim2_iter_value = sequencer_iter[op_instr.dim2_iter_id].count[7:0]
dim3_iter_value = sequencer_iter[op_instr.dim3_iter_id].count[7:0]

# compute iterator * stride for each dimension.
# In case of overflow, only the 12 LSBits are kept for each product
dim1_prod[11:0] = op_instr.hbuf_stride_dim1[11:0] * dim1_iter_value[7:0]
dim2_prod[11:0] = op_instr.hbuf_stride_dim2[11:0] * dim2_iter_value[7:0]
dim3_prod[11:0] = op_instr.hbuf_stride_dim3[11:0] * dim3_iter_value[7:0]

#compute sum of hbuf_base, per-instruction fixed offset, and per-dimension offsets
hbuf_addr[11:0] =
    ({hbuf_base[11:4], 4'b0}  +  #  hbuf_base address maintained by hardware
     op_instr.hbuf_offset[11:0] +
     dim1_prod[11:0] +
     dim2_prod[11:0] +
     dim3_prod[11:0]) % 4096  # modulo hbuf_size, so address can wrap


# Break computed address into hbuf bank, word-in-bank, and byte offset, to WDP:
hbuf_addr_in_bank = hbuf_addr[11:4]
hbuf_bank         = hbuf_addr[3]
hbuf_byte_offset  = hbuf_addr[2:0]
```

10.1.2 Hbuf FIFO Read Operation

The Hbuf memories operate as FIFOs, where the number of entries is incremented and decremented in chunks of **hbuf_block_size**. The current occupancy of the Hbuf is maintained in the hbuf_rd_credits register by hardware.

Periodically, the hbuf_base address is advanced by the hbuf_block_size, under program control, when an entire "block" of data has been read out of hbuf memories. At that point, hbuf_block_size is also decremented from the hbuf_rd_credits register, which is effectively a return of that many credits to the WDP Sequencer (which writes the Hbuf FIFOs). The hbuf_base address wraps around the hbuf memory size automatically.

For example, an hbuf_block might be configured for 1024 bytes (per grid-row). The WDP Sequencer will not increment the hbuf_rd_credits until it has written 1024 bytes to every grid-row. Once the hbuf_rd_credits is nonzero, the GH Read sequencer can safely read 1024 bytes, since by policy, that would be the agreed-upon (programmed) hbuf_block_size. After consuming that data, GH Read sequencer signals a decrement (by the block size, i.e, 1024) back to WDP Sequencer.

```
# maintain hbuf_base and credits
if op_instr.hbuf_block_start and op_instr.hbuf_block_iter_mask != 0:
    block_start = True  # init
    for i in range(num_iters):
        if (op_instr.hbuf_block_iter_mask[i]) and iter[i].value != 0:
            block_start = False
    if block_start:
        stall if hbuf_read_credits == 0   # wait for WDP to send credits

if op_instr.hbuf_block_end and op_instr.hbuf_block_iter_mask != 0:
    block_done = true  # init
    for i in range(num_iters):
        if (op_instr.hbuf_block_iter_mask[i]) and !iters_eq_nloops[i]:
            block_done = False

    if block_done:
        hbuf_base += op_instr.hbuf_block_size
        decrement hbuf_read_credits by hbuf_block_size
        increment WDP sequencer hbuf_write_credits by hbuf_block_size
```

10.1.3 GH Sequencer Algorithm for row-shifted MatMul

```
# GH sequencer for row-shifted MatMul (FP8):
# Hbuf block size = 1024B (per grid-row)

for hblock_iter in range (hbuf_blocks):
    for octobyte_in_word_iter in range (16):        # 16 octobytes in each word
        for virtual_grid_row_iter in in range (8):  # 8 words per grid row
            read_hbuf(
                hbuf_rd_cmd = RD_1X1_MATMUL_FP8,
                hbuf_block_size = 64,               # *16 = 1024bytes
                hbuf_block_start_en = 1,
                hbuf_block_end_en = 1,

                hbuf_block_iter_mask =
                    MASK(octobyte_in_word_iter, virtual_grid_row_iter)
                hbuf_stride_dim1 = 8,               # octobyte stride
                hbuf_stride_dim2 = 128,             # virtual_grid_row stride
                hbuf_stride_dim1_iter_id = octobyte_in_word_iter,
                hbuf_stride_dim2_iter_id = virtual_grid_row_iter)
```

Figure 10-1: GH Seq pseudo-code for "row-shifted" Matmul (FP8 and FP16)

For **FP16 row-shifted MatMul**, the GH sequencer algorithm is the same as for FP8, except the hbuf_rd_cmd used is RD_1x1_MATMUL_FP16

10.1.4 GH Sequencer Algorithm for transposed Matmul

```
# GH sequencer for transposed FP8:
# Hbuf block size = 64B (per grid-row)

for hblock_iter in range (hbuf_blocks):
    for hbuf_word_in_bank_iter in range (4):     # word-in-bank per hbuf block
        for quadbyte_iter in range (2):          # 2 quadbytes in each word
            read_hbuf(
                hbuf_rd_cmd = RD_TRANS_1X1_MATMUL_FP8,
                hbuf_block_size = 4,             # *16 = 64bytes
                hbuf_block_start_en = 1,
                hbuf_block_end_en = 1,
                hbuf_block_iter_mask = MASK(hbuf_word_in_bank_iter, quadbyte_iter)
                hbuf_stride_dim1 = 4,            # quad-byte stride
                hbuf_stride_dim2 = 16,           # word-in-bank stride
                hbuf_stride_dim1_iter_id = quadbyte_iter,
                hbuf_stride_dim2_iter_id = hbuf_word_in_bank_iter)
```

Figure 10-2: GH Seq pseudo-code for FP8 Transposed Matmul

```
# GH sequencer for transposed FP16:
# Hbuf block size = 64B (per grid-row) which corresponds to 8 AMEM data words (1024B)

for hblock_iter in range (hbuf_blocks):
    for quadbyte_iter in range (2):                    # 2 quadbytes in each word
        for hbuf_word_in_bank_iter in range (4):       # word-in-bank per hbuf block
            read_hbuf(
                hbuf_rd_cmd = RD_TRANS_1X1_MATMUL_FP16,
                hbuf_block_size = 4,             # *16 = 64bytes
                hbuf_block_start_en = 1,
                hbuf_block_end_en = 1,
                hbuf_block_iter_mask = MASK(hbuf_word_in_bank_iter, quadbyte_iter)
                hbuf_stride_dim1 = 4,   # quad-byte stride
                hbuf_stride_dim2 = 16,  # word-in-bank stride
                hbuf_stride_dim1_iter_id = quadbyte_iter,
                hbuf_stride_dim2_iter_id = hbuf_word_in_bank_iter)
```

Figure 10-3: GH Seq pseudo-code for FP8 Transposed Matmul

10.1.5 GH Sequencer Algorithm for 3x3 conv

TBD

10.2 GV (GRID VERTICAL DATA) SEQUENCER

GV Sequencer Op instruction format:

Field	Width	Description
opcd	2	0: NOP 1: read zero data (not from a source FIFO) 2: read from AMEM-source switchbox FIFO (if nonempty), and pop that FIFO 3: read from VPU-source switchbox FIFO (if nonempty), and pop that FIFO
conv3x3_mode	1	0: 1x1 or matmul, send v_data to v_staging back buffer 1: 3x3 conv, send v_data to v_staging front buffer (3-cycle staging pipe) The mode (1x1 vs 3x3) may not be changed within a single trip. All instructions of a trip must have the same value for this mode
staging_start_iter_mask	6	When all iterators indicated by the mask have value=0, signals “staging start” for the current data. This starts the 8-entry sequence for writing the v_staging back buffer. Enabled for matmul/1x1 mode only When the mask is all zero, disables “staging start”
lin2log dsbl_mapping_corr	1	Disable the mapping correction in the Lin2log conversion on the way into the grid. This can be used for an identity round-trip (lin-log-lin).
lin2log fbits_truncate_amt	3	Number of Fraction bits to truncate (force to zeros) in the LnsI5F10 value after Lin-to-Log mapping correction. Useful for maping FP8 to LnsI5F10 with excess precision in the LNS value; masking out some LSBs can save power in the LLC Grid without impacting accuracy. To keep all 10 Fbits, set this config to 0x0; to reduce to 3 F bits (followed by 7 zero-bits) set this to 0x7.

		Should be set to 0 for FP16 inputs to keep all 10 Fbits in the LnsI5F10 value.
lin2log eb_adj	6	signed 6-bit integer exponent adjustment value to apply in lin2log converter

10.3 GX (GRID EXECUTION) SEQUENCER

GX Sequencer Op instruction format:

Field	Width	Description
opcd	2	0=Exec NOP 1=Exec Valid, 2=Exec Bubble 3=Exec Config Exec NOP does not require (nor consume) any h_data or v_data. This can be used for a single no-op trip when the grid is not being used. Exec Valid is a normal execution cycle in the grid. For 1x1/matmul, this requires h_data to be valid to enable execution. For 3x3, this requires v_data source (as selected by the v_data sequencer) to have valid data to enable execution Exec Bubble instructs the sequencer to disable execution in the LLCs, but still requires v_data and h_data to be available, and "consumes" the h/v data This is primarly needed for 3x3 convolution, to fill the vertical staging buffers at the start of a group of 8 of out 10 execution cycles, in which the first two cycles are execution bubbles. Note this is distinct from the No Op Opcode because it requires incoming data & weights to be valid in the LLC staging buffers. Exec Config is used to apply a grid execution configuration at the start of a trip. It does not require (nor consume) any h_data or v_data, and does not perform any computations in the LLCs. The configuration will remain active until changed by another Exec Config instruction.
conv3x3_mode	1	0: 1x1 or matmul 1: 3x3 convolution The mode (1x1 vs 3x3) must not be changed within a single trip. All instructions of a trip must have the same value for this mode
Grid Execution fields (applicable for opcd = ExecValid or ExecBubble)		
accum_idx_iter_id	3	which iterator is used to drive the accumulator index (read & write, to active accumulator)
front_staging_done iter_mask	6	When all iterators indicated by the mask have their ending loop value, this indicates that the appropriate staging "front" buffer is done and can be reloaded from its companion "back" buffer. **For 1x1/matmul, this is the v_data front buffer; for 3x3, this is the h_data front buffer**. When the mask is zero, staging-done is never indicated
zero_accum_iter_mask	6	When all iterators indicated by the mask have value=0, this indicates the first accumulator-group after a writeback or split accumulation, and this

		instructs the accumulator read hardware to return zeros to effectively clear the accumulators at the beginning of accumulation. When the mask is zero, the active accumulator is not forced to return zeros
split_accum_iter_mask	6	When all iterators indicated by the mask have their ending loop value, the LLCs will trigger a "split" accumulation. This requires that any prior Writeback (offloading the Writeback accumulators) is complete. As part of a split accumulation, all of the *active accumulator* (AA) slots are automatically read and summed with the *writeback accumulators (*WBA*)*, with the results stored back into the WBA. Hardware will automatically return zeros for the WBA read-data on the first (or only) split after a writeback. After a split, the next set of (normal execution) reads from the AA must return zeros (under control of **zero_accum_iter_mask**), and permit the split processing to "steal" the AA read port. Every accumulation must have at least one split to move data to the WBA. When the mask is zero, split accumulation is not signaled
wb_kick_iter_mask	6	When all iterators indicated by the mask have their ending loop value, the next (or only) split accumulation will also kick the Grid Writeback sequencer to offload the Writeback accumulators from the grid back to the Switchbox. This only takes effect on the instruction in which a split accumulation is also being triggered by the **split_accum_iter_mask** It is expected that bits set in the **split_accum_iter_mask** is a subset of the **wb_kick_iter_mask**, as all cases of WB kick coincide with triggering a split, but not all splits coincide with WB kicks.
llc_dsbl_mapping_corr	1	Can be set to disable the log-to-linear mapping correction (within the LLCs). Needed for identity round-trip operation
Grid configuration fields (applicable for opcd=ExecConfig)		
end_grid_row_idx	4	Supports a subset of active grid-rows, numbered from grid-row 0 through **end_grid_row_idx** inclusive. This is for partial grid execution, such as when then number of output channels is smaller than the number of grid-rows. The LLCs in inactive rows are not enabled to compute results, and their accumulator values will be undefined. For normal (full grid) operation, set this to 15.
log2_active_ptns	2	Supports 1, 2, 4 or all 8 partitions active, always starting from partition 0. The LLCs in inactive columns are not enabled to compute results, and their accumulator values will be undefined. This supports partial grid execution when the tensor width only requires a subset of the columns enabled each cycle. (This is useful for power saving with matrix*vector calculations)

log2_ptns_per_filter	2	1, 2, 4 or 8 partitions per filter (on each grid-row); applicable for 3x3 mode only, for narrow tensors where multiple filters (output channels) may be mapped to the same grid row. Needed by hardware to determine tensor boundaries to "break" sharing of columns for 3x3 conv. This field is a don't-care for 1x1 conv or MatMul
odd_col_exec_en	1	enables LLC execution for odd-numbered columns. May be disabled for stride-2 convolutions with alignment to even columns
even_col_exec_en	1	enables LLC execution for even-numbered columns. May be disabled for stride-2 convolutions with alignment to odd columns

Decoding the execution-configuration parameters:

```
# Find partition-edges on power-of-two-partition group boundaries.
n_ptns_per_filter = 1 << op_instr.log2_ptns_per_filter
for ptn in range (8):
    ptn_cfg[ptn].ptn_edge_west = (ptn % n_ptns_per_filter) == 0
    ptn_cfg[ptn].ptn_edge_east = (ptn % n_ptns_per_filter) == n_ptns_per_filter- 1;

# Combine even/odd exec en with active_ptns
for ptn in range (8):
    if (ptn >= (1 << op_instr.log2_active_ptns)):
        # partition not active
        ptn_cfg[ptn].even_col_exec_en = False
        ptn_cfg[ptn].odd_col_exec_en  = False
    else:
        ptn_cfg[ptn].even_col_exec_en = op_instr.even_col_exec_en
        ptn_cfg[ptn].odd_col_exec_en  = op_instr.odd_col_exec_en
```

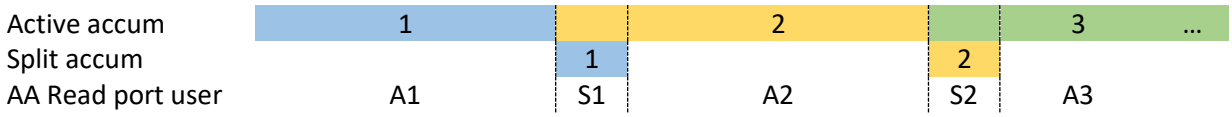
10.3.1 Active - Split Accumulation Overlap

Each LLC has 8-slot Active Accumulator (AA) and 8-slot Writeback Accumulator (WBA) storage. "Split accumulation" requires reading values from one slot in the AA and WBA, summing, and storing the sum in WBA at that slot. This is repeated for each of the 8 slots.

After a split is computed, the LLC starts a new accumulation in the AA, which means the next 8 (active) computations will treat the AA value as zero. This is true at the start of every accumulation as well, and is controlled by the **zero_accum_iter_mask.**

This zeroing of the AA allows the active LLC computation and split accumulation to *share* the AA read port and proceed in parallel. When a split accumulation is initiated (as controlled by the **split_accum_iter_mask**), hardware will perform the 8-cycle split accumulation operation, and this may happen *concurrently* with the start of the next set of active accumulations. This requires that the next set of (at least) eight active LLC computations ***must not*** be using the AA read port – so **zero_accum_iter_mask *must*** be set to zero-out the AA reads at the start of active accumulation following each split.

If the split accumulation is stalled (discussed in the next section), then this will also stall the next active accumulation – cycle for cycle – to ensure that the split can finish its 8-cycle usage of the AA read port before the active LLC computation needs to read from AA port.

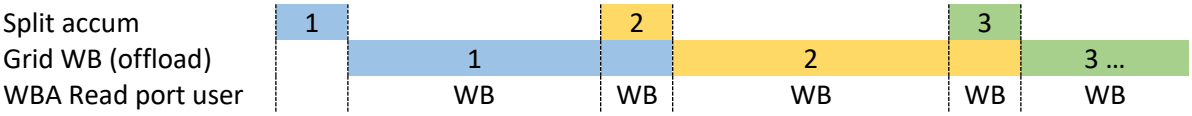


10.3.2 Split Accumulation – Writeback Overlap

When dot-product computation is completed in the grid, the results are always stored into the Writeback Accumulator (WBA) and then off-loaded by the Grid Writeback (GWB) sequencer which reads each slot in the WBA in each grid-row in turn.

After the WBA is off-loaded, the WBA can be used for a new set of computations. The first split after a writeback will automatically treat the WBA value as zero for reads, so that split accumulation will perform the operation `WBA[slot] <- AA[slot] + 0`. (This is managed by hardware, not under microcode control.) This allows the split to *share* the WBA read port with the GWB off-loading, and permits the first split to proceed in parallel with the end of the prior GWB off-load.

It is always necessary to perform at least one split accumulation to *move* the results from the Active Accumulator to WBA, prior to offloading the results from the WBA. If this is the only split, then it is effectively only reading the AA and writing the WBA, and does not need to use the WBA read-port. The following figure shows overlap of split and writeback offload with a single split



Although the first split after writeback does not need to use the WBA read-port, it will overwrite the WBA contents, and this must not race ahead of those contents being offloaded by the *prior* writeback. In the figure above, this is a hazard for split accum 2 overlapping the end of WB 1. Hardware ensures that the first split can only overlap the tail end of each writeback offload, so data is overwritten safely behind the offload read.

In particular, the split accumulation is allowed to write to WBA slot 0 (in *all* grid-rows) just after GWB has read (off-loaded) slot 0 in the *last* grid-row, and may write slot 1 (in all grid-rows) just after off-loading slot 1 in the last grid-row, etc.

This uses *Writeback Credits* from GWB to GX. GX (split accumulation) starts with 8 credits, consumed for each slot written by a split accumulation which will initiate a writeback offload. GWB then returns the writeback credits when it is offloading accumulators from the *last* grid-row (one credit per accumulator slot offloaded). The credits allow split accumulation to overwrite the WBA slots just after GWB has read the same slot. If the next split does not have a credit, it is stalled.

Without this optimization, the split accumulation would have to wait until all 8 slots are offloaded before overwriting the first slot, costing 7 cycles execution delay. This 7 cycle delay is meaningful for short-computation tasks, such as for transformer Q*KT calculations with ~128 features per attention-head, where the computation time per tile is only 128 cycles. The writeback offload time is also 128 cycles, so this technique saves stalling for 7 out of 128 cycles.

In case of multiple split accumulations between writebacks, only the first split must potentially be stalled. Subsequent split accumulations would never be in potential conflict with the prior writeback offload.

11 GRID WRITEBACK SEQUENCER

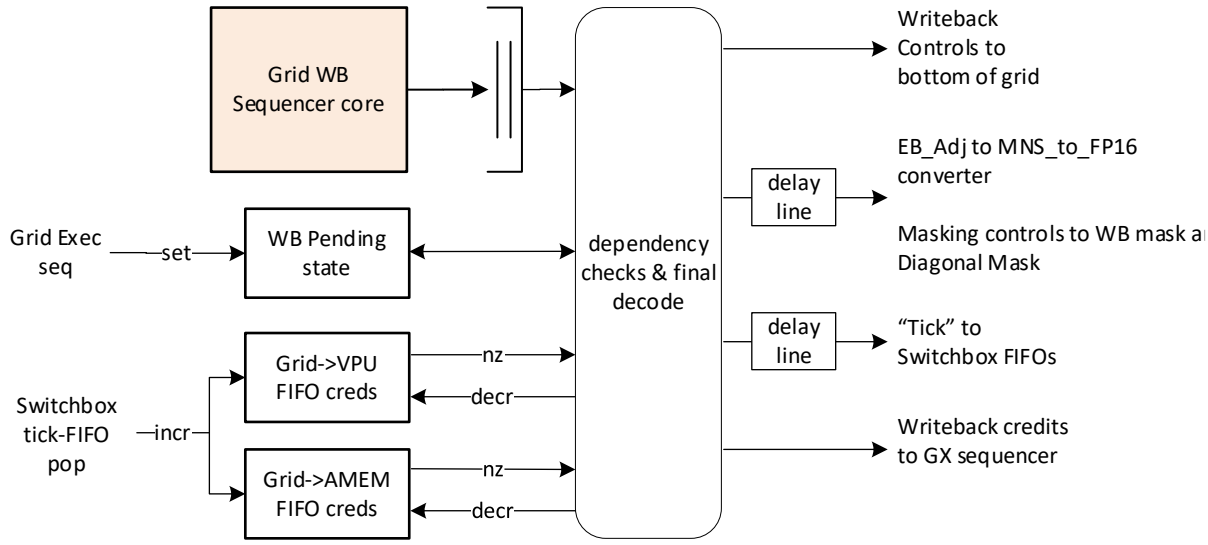


Figure 11-1: Grid Writeback Sequencer

The Grid Writeback (GWB) Sequencer controls "writeback" or offloading from the Grid writeback accumulators to the switchbox FIFOs. This sequencer also provides the EB adjustment value for the MNS-to-FP16 conversion, and masking logic which is applied just as data exits the Grid. This sequencer is conceptually located at the "bottom" of the Grid, sending controls flowing "up" from bottom to top of the grid.

Each cycle GWB can issue a read command targeting one Writeback Accumulator (WBA) slot in one grid-row. It loops over all 8 WBA slots for one grid-row, then all 8 for the next grid-row, etc, under program control. The selected accumulator slot data is forwarded to the MNS-to-FP16 converter, and then to either the AMEM-facing or VPU-facing FIFO in the switchbox.

A typical microprogram for GWB iterates over all 16 grid-rows x 8 WBA entries to offload one *tile* for 1x1/linear/Matmul layers. It may further iterate over multiple tiles in a single trip. For 3x3 conv, GWB may be programmed to read the WBA slots from each grid-row multiple times for "narrow tensor" cases where multiple output channels share the same grid-row.

GWB also supports mixed EB conversions within a single tile. This may occur for some "outlier" channels. The sequencer allows for dividing the writeback offloading (for each tile) into subsets of contiguous grid-rows, where each subset has a unique EB Adjustment amount. The granularity is 8 WBA slots always. For example, it is possible to read 15 grid-rows x 8 WBA slots (120 output channels) with EB_Adj X, then 1 grid-row x 8 WBA slots (8 output channels) with EB_Adj Y. This example may be programmed with two microinstructions, one covering the first 15 grid-rows, and the 2nd covers a final grid-row.

Note: If not all grid-rows are in use for a trip, the active grid-rows always start from grid-row index 0.

11.1 GWB SEQUENCER OP INSTRUCTION FORMAT

Field	Width	Description
opcd	1	0: NOP 1: read 8 slots from WriteBack Accumulator for the selected grid-row
tgt_fifo	1	0: send data to AMEM-facing switchbox FIFO 1: send data to VPU-facing switchbox FIFO
diagonal_mask_mode	3	Determines when/how values are replaced with **diagonal_mask_override_sel** in the 128x128 output tile matrix: 0: diagonal masking disabled 1: replace if col <= row 2: replace if col < row 3: replace if col == row 4: replace if col != row 5: replace if col >= row 6: replace if col > row
diagonal_mask override_sel	1	If a data element is selected for masking by the **diagonal_mask_mode**, replace with the following value: 0: replace data with **0** 1: replace with **max_negative** (fp16)
col_mask_idx	2	Selects one of four column-mask registers. Each mask register provides per-column-configurable **override** masking, with options for (no-mask, zero, max_negative) for each of the 128 columns of data.
eb_adj	6	signed 6-bit integer exponent adjustment value to apply in MNS2FP converter
grid_row_offset	4	Starting grid-row index
grid_row_iter_id	3	Determines which iterator selects the grid-row index
wb_done_iter_mask	6	When all iterators indicated by the mask have their ending loop value, this indicates that the writeback is done for this tile. The GWB sequencer will then wait for the next “wb_kick” from the grid-exec sequencer. When the mask is zero, writeback-done is not indicated. The last (non-NOP) instruction for each tile MUST set this mask, and only one instruction may set this mask (nonzero) per tile written back

Grid Writeback (offload) is initiated by the writeback-kick received from the Grid Exec sequencer, and may be programmed for multiple writeback operations per trip (equally on both Grid Exec and Grid WB sequencers).

The sequencer’s primitive operation is to read 8 accumulator slots from a programmed grid-row. In a typical case, the sequencer will be programmed to loop through all grid-rows (starting from grid-row 0) reading and offloading 8 accumulators each.

When the grid_row iterator reaches its ending loop value, it can be programmed to trigger hardware to return Writeback Credits from GWB to GX. These credits are described below, and in section 10.3.2.

The sequencer also provides control points for the writeback datapath elements from the Grid to the Switchbox.

11.2 Column-masking and Diagonal Masking

The Grid-writeback datapath provides two types of masking on the data, on writeback (offload) from the grid to the switchbox:

- "diagonal" masking, used for overriding values on diagonals or lower-left or upper-right matrix triangles. The sequencer program specifies which of the types of masking to apply, and when overriding the data values, whether to replace with zero or -max
- "column" masking, used for overriding individual columns of data. The sequencer program specifies which of 4 column-mask registers to apply. Each column-mask register may be (preconfigured) with per-column override values to indicate (no-override, force zero, or force -max).

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.

11.3 GWB Grid-Row Index calculation

GWB microcode allows unrolling loops to access subsets of contiguous grid-rows. The effective grid-row index is the sum of the `grid_row_offset[3:0] + iterator<grid_row_iter_id>.count[3:0]`. The calculated value must be within the legal range 0..15 inclusive, and must not overflow or loop around – this is a programming requirement (not enforced by hardware).

11.4 GWB Sequencer Flow Control

GWB operation requires handshaking with the Grid Execution (GX) sequencer for each tile. When a tile's computations are complete and need to be offloaded, GX issues a writeback "kick" which sets a WriteBack Pending state bit for GWB. GX may not issue another writeback kick until the previous writeback is complete, and more strictly, a GX Split accumulation may not be executed until the prior WB is complete (at which point the WriteBack Accumulators are free to receive new data).

Hardware provides an optimization to allow partial overlap of a writeback (by GWB) with the next split accumulation (in GX) using *Writeback Credits* from GWB to GX. See section 10.3.2 for more details on use of these credits. GWB returns writeback credits near the end of each writeback, when it is offloading accumulators from the last grid-row (one credit per accumulator slot offloaded). This is controlled by the **wb_done_iter_mask**.

GWB operation also requires handshaking with the destination switchbox FIFOs, using traditional FIFO credits. GWB maintains credits for each target FIFO in the switchbox. A credit is decremented from the appropriate FIFO for each flit (i.e, per GWB read command to the grid). Credits are returned as each FIFO is popped at the switchbox.

12 VPU INSTRUCTION SEQUENCER

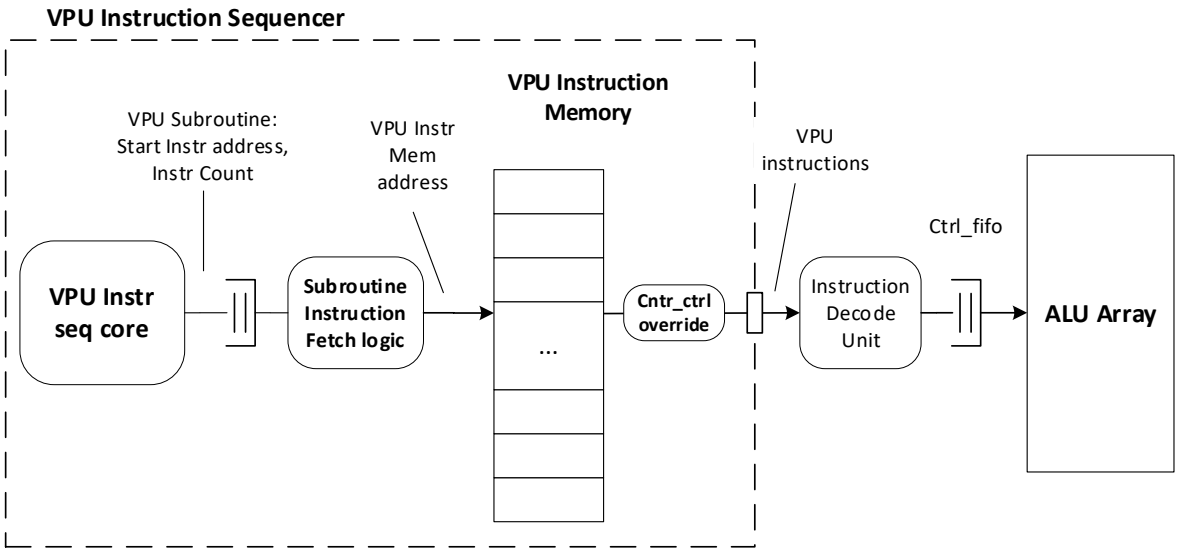


Figure 12-1: VPU Instruction Sequencer

The VPU Instruction (VI) Sequencer generates the series of VPU instructions to be executed for each trip. This is a two-phase sequencer:

- The first phase is a conventional microcoded sequencer core which generates a series of VPU "subroutine" commands. Each command includes a starting VPU program address, and instruction count to reference the VPU Instruction Memory.
- The second phase fetches the VPU instructions from VPU Instruction memory, as specified by the starting address and number of instructions. The stream of VPU instructions is then delivered to the VPU Instruction Decode Unit (IDU).

Note that this sequencer has two distinct programmable memories: the sequencer core microcode, and the VPU Instruction Memory. The latter is unique to this sequencer.

Each processing phase is decoupled with a FIFO: from the sequencer core (first phase) to the VPU program instruction fetch (2nd phase), and from the fetched VPU instructions to ALU control FIFO.

A simple VPU program may consist of a single VPU instruction executed for every tensor row of a tile streamed through the VPU. For this case, the sequencer would be programmed to execute the 1-instruction program repeatedly in a loop. For more general cases, the sequencer core (first phase) can be programmed to execute multiple subroutines or program-fragments, looping on these as needed.

Each subroutine or program-fragment may consist of up to 1024 instructions. The VPU Instruction Memory provides storage for 1K VPU instructions. This memory is software managed.

12.1 VPU INSTRUCTION SEQUENCER OP INSTRUCTION FORMAT

VPU Instruction Sequencer Op instruction format:

Field	Width	Description
opcd	1	0: NOP 1: Generate VPU Program
vpu_pgm_start_addr	10	Starting VPU-Program address intp the VPU Instruction Memory, to execute a program or program-fragment, consisting of sequential VPU instructions from that memory
vpu_pgm_instr_count	10	Number of VPU instructions to execute Encoded as “N-1” so codepoint 0=1 instruction, 1=2 instructions, etc.
counter_zero_iter_mask	6	This control is used to reset the VPU’s Counter Register at the end of a (nested) loop. When all iterators indicated by the mask have their ending loop value, the sequencer will force the **cntr_ctrl** field = **CNTR_CTRL_SET_TO_0** in the **last** instruction in the generated VPU program. This effectively resets the counter register at the end of the last VPU instruction in the last loop iteration as programmed. This is useful for looping on subroutines which process a large tensor, where the subroutine would have (at least) one instruction which increments the VPU Counter register. This feature avoids unrolling the last loop with extra VPU program code to reset the Counter. When the mask is zero, the override is never enabled.

13 VBUFFER CONTROL

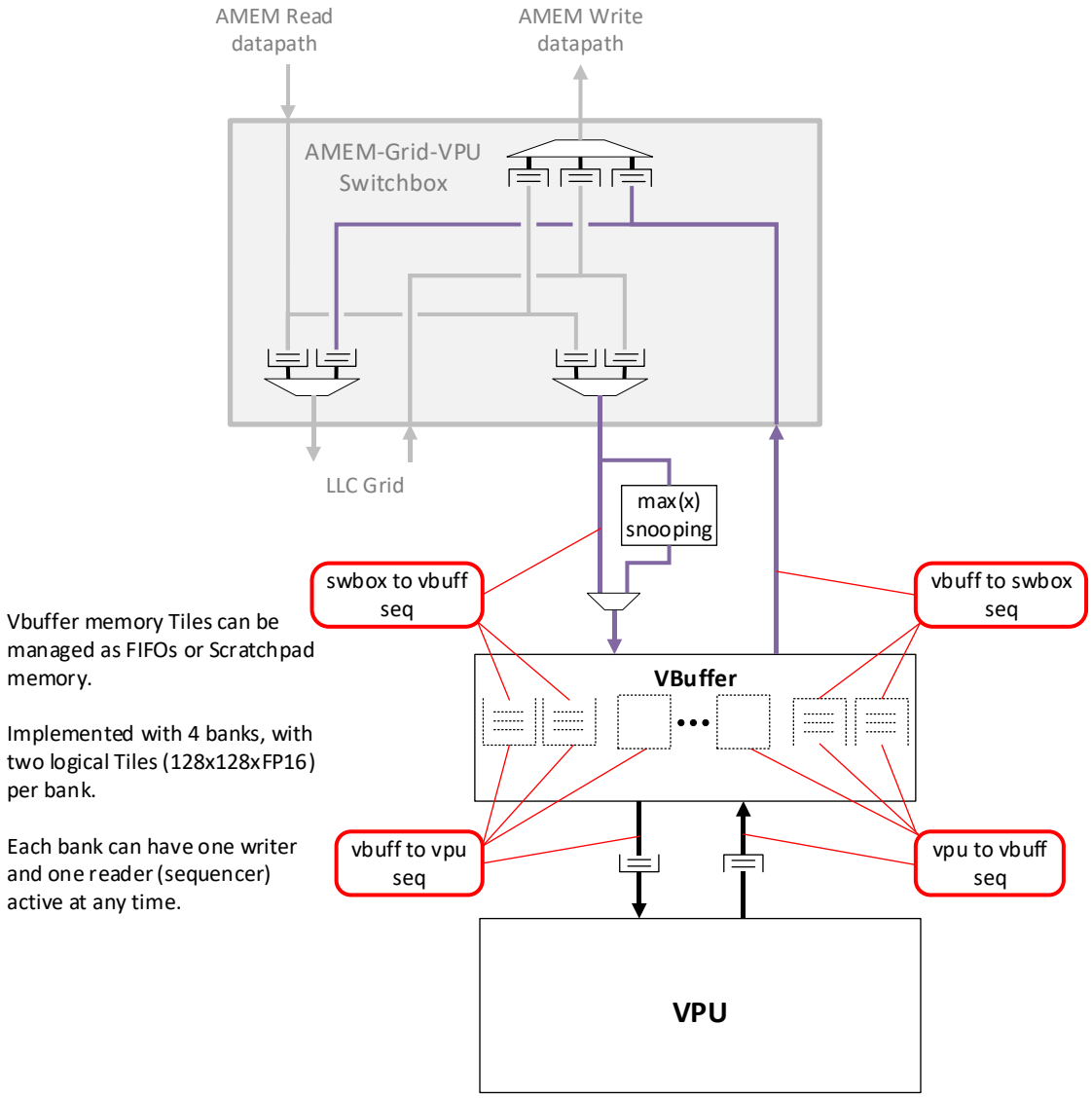


Figure 13-1: Vbuffer Sequencers overview

There are four similar sequencers to control movement of data into and out of the **Vbuffer** memories:

- **Switchbox-to-Vbuff** sequencer reads data from a Switchbox FIFO and writes to a Vbuffer tile
- **Vbuff-to-Switchbox** sequencer reads data from a Vbuffer tile and writes to a Switchbox FIFO
- **Vbuff-to-VPU** sequencer reads data from a Vbuffer tile and writes to the VPU InFIFO
- **VPU-to-Vbuff** sequencer reads data from the VPU OutFIFO and writes to a Vbuffer tile

13.1 VBUFF TILE FIFO OPERATION

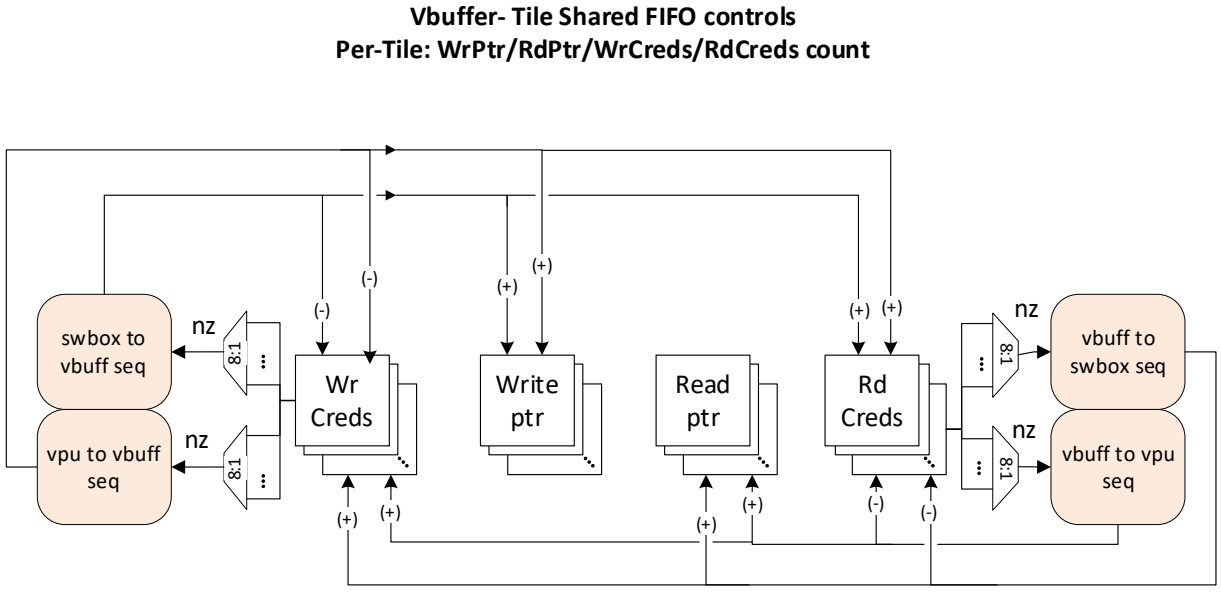


Figure 13-2: Shared Vbuff Tile FIFO controls

The Vbuffer tiles can each be configured to operate as FIFOs, and the sequencers share the FIFO read & write pointers for each of the tiles. At any time, either the Switchbox-to-Vbuff sequencer or VPU-to-Vbuff sequencer may write to a tile FIFO and control the tile's FIFO Write pointer. Similarly, at any time either the Vbuff-to-Switchbox sequencer or Vbuff-to-VPU sequencer may read from a tile FIFO, and control the tile's FIFO Read pointer. For flow control, the controlling sequencers will also share FIFO credits and fill status. This will naturally regulate the sequencer operation rate, so the Vbuff-writing sequencers cannot push to a full FIFO, and Vbuff-reading sequencers cannot pop from an empty FIFO.

13.2 VBUFF TILE SCRATCHPAD OPERATION

The Vbuffer tiles can also be configured to operate in "scratchpad" mode by the sequencers, selected on a per-tile basis. In this mode, the sequencers can make random-access reads or writes to a tile, where the sequencer microprogram directly computes the address into the tile memory (rather than using FIFO pointers).

This mode allows the Vbuffer tiles to be used as cache-like storage. For example, the VPU can offload a set of registers and fetch them back later, or constants can be written to a Vbuffer tile and read back multiple times.

In Scratchpad mode, the sequencers use a special flow control & dependency resolution mechanism to ensure that a Vbuff-reading sequencer does not access an entry (in a tile memory) before it has been written by the associated Vbuff-writing sequencer. This "Read After Write" (RAW) hazard exists because there is a potentially indefinite delay in writing into the Vbuffer, which could allow the Vbuff-reading sequencer to race ahead of the Vbuff write.

There is **no** equivalent protection provided from "Write after Read" (WAR) hazards. This hazard *could* arise if a Vbuff-writing sequencer could overwrite Vbuffer (scratchpad) entries before they have been read and used. In most cases, the FIFO mode is better to prevent overwriting un-read entries in the Vbuffer. For the case of VPU writing values and reading them back later, the WAR hazard is impossible because a read followed by a write in the VPU program will always complete the read before the write can be issued. So, this protects the VPU2Vbuff sequencer from clobbering (overwriting) Vbuff scratchpad data before the Vbuff2VPU sequencer has read the data. For other sequencer writer/reader combinations, it is the programmers responsibility to ensure the WAR hazard is impossible.

For RAW protection, hardware provides a (per-tile) up/down **scratchpad_count** register to indicate the number of entries which have been written and may be safely read. This count is incremented by the Vbuff-writing sequencer, and decremented by the Vbuff-reading sequencer under microcode control.

Software sets up the increments on Vbuff-writing Sequencer microcode instructions, and decrements on Read instructions, with a knowledge of the interoperation of the read & write programs. This is critical if the variables are read in a different order than they are written. A Vbuff-write operation can increment the **scratchpad_count** by 'N' to indicate that the Vbuff-reading sequencer may safely read up to N values from the tile memory. A Read operation can only proceed if the **scratchpad_count** is greater than zero, and will decrement the count the first time it reads each memory location (which should usually be only once per tile memory location).

For example, consider writing and reading three variables into a tile at memory locations A, B, & C:

- If the write order is (A,B,C) and read order is (A,B,C), then each Write operation will increment the count by one, and each Read operation will decrement by one.
- If the write order is (A,B,C) and read order is (C,B,A), then the Vbuf-writing sequencer will not increment the count until it writes C, at which point it would increment the count by 3. This blocks the Vbuf-reading sequencer from reading C too early. The Reads would again decrement the count with each read. (These behaviors must be programmed into sequencer microcode).

If there are additional blocks of write/read variables, the count can again be incremented and decremented as many times as necessary. The maximum increment amount is 128, and the maximum value that the **scratchpad_count** register can ever have is 128 (the tile depth). At the end of a trip, the count **must always return to zero**. Software is responsible for matching the increments and decrements programmed into the sequencer microcode.

For spilling/filling blocks of registers in a loop, the count can be incremented by N (the number of values written in the loop) on the last loop iteration, and later decremented by 1 for every value (every loop iteration). This allows emulating a spill/fill stack. For example, suppose 16 VPU GPR values are written to memory locations 0-15, and then later, another set of registers are written to memory locations 16-31. If the program first reads back locations 16-31, and then later reads back 0-15, then the Vbuf-writing sequencer would be programmed to increment the count by 32 when it completes writing the last value to location 31.

13.3 VBUFF BANK AND TILE SHARING RULES

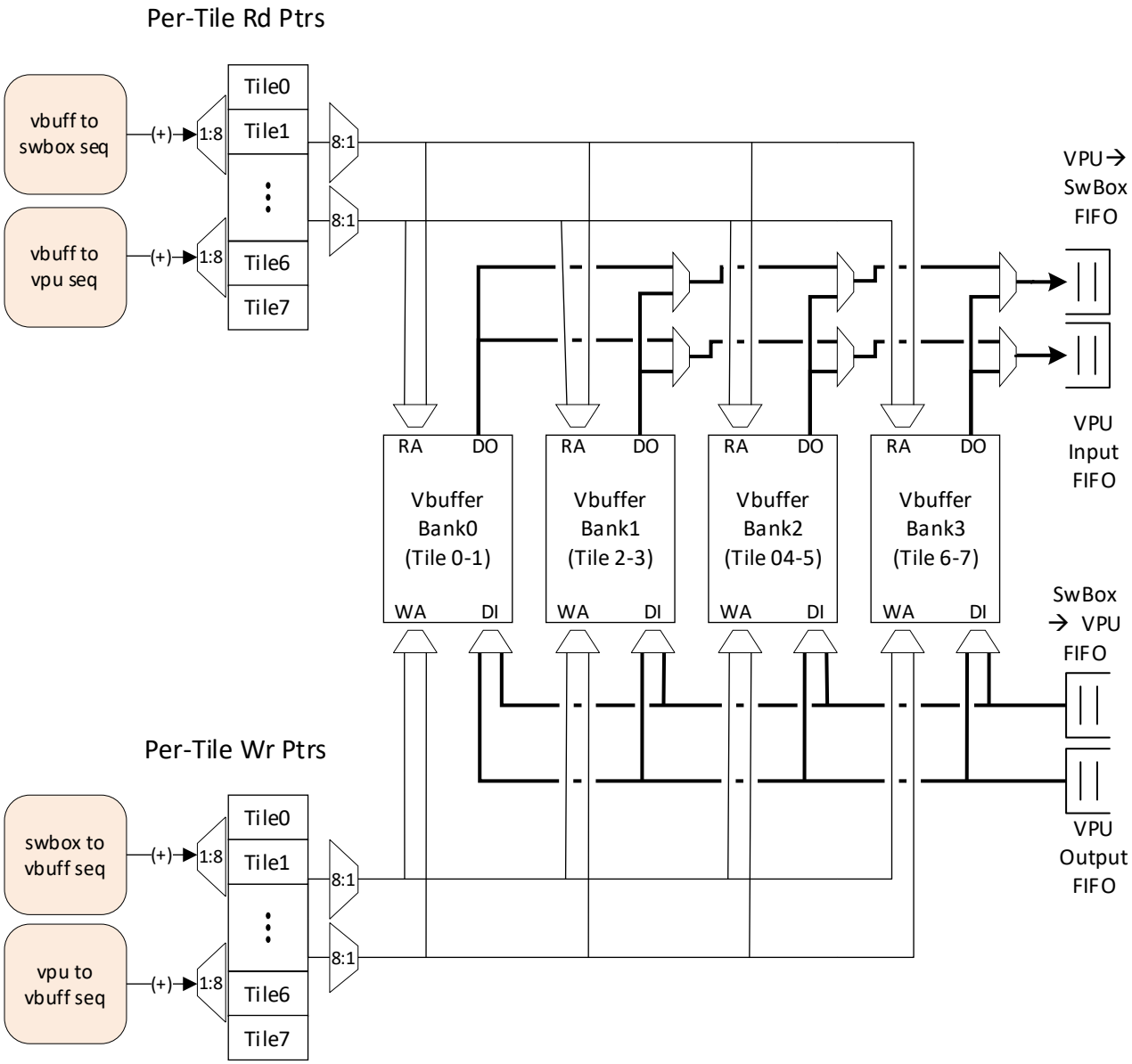


Figure 13-3: Shared Vbuff Bank & Tile access for four Vbuff sequencers

The Vbuffer memories are organized as 4 banks of two-ported memories, where each bank has two logical tiles (128x128 FP16 entries per tile).

- Each bank can have only one vbuff-writing sequencer (swbox-to-vbuff or vpu-to-vbuff) which “owns” the write access to that bank – and both tiles in the bank – at any one time, typically for the duration of a trip.

- Similarly, each bank can have only one vbuff-reading sequencer (vbuff-to-swbox or vbuff-to-vpu) which “owns” the read access to the bank and both of its logical tiles at any one time, typically for the duration of a trip.

Each sequencer is programmed with the bank(s) and tile(s) it needs to use, and it is up to software to adhere to these rules to avoid hardware conflicts in accessing the banks. If the sequencers’ ownership of Vbuff banks needs to be changed, a barrier may be used to force the sequencers to be completely idle and the banks to be empty before a trip using different sequencer-to-bank ownership can be enabled.

14 VBUFFER SEQUENCERS

14.1 SWITCHBOX-TO-VBUFF SEQUENCER (SB2VB)

This sequencer moves data from the Switchbox FIFOs or a "snooping" register to the Vbuffer memories. It also controls the snooping logic which finds the maximum value in a stream of data being written into the Vbuff.

Data may be written using FIFO mode semantics, or an address to a tile may be directly computed (in Scratchpad mode) from a base address and an iterator value.

SB2VB Sequencer Op instruction format:

Field	Width	Description
opcd	1	0: Nop 1: WR_VLD. Move data from a source FIFO or snooping-register into the Vbuff.
wr_tile_idx	3	Selects which Vbuff Tile will be written to (managing the tile as a FIFO)
src_select	2	0: read data from AMEM-facing switchbox FIFO 1: read data from Grid-facing switchbox FIFO 2: read data from max-value snooping register
scratchpad_mode	1	0: the sequencer writes to the Vbuff in FIFO mode 1: the sequencer writes to the Vbuff in "scratchpad" mode
Scratchpad Mode controls		
scratchpad_count_iter_mask	6	When all iterators indicated by the mask have their ending loop value, **or** if the mask is all-zero, the **scratchpad_count** register (for the tile indicated by **wr_tile_idx**) is incremented by the **scratchpad_count_incr_amt.**
scratchpad_count_incr_amt	8	Amount to increment the **scratchpad_count** register (if enabled by the **scratchpad_count_iter_mask**) Maximum useful value = 128 (size of one tile)
wr_addr_base	7	Base address for random-access into the selected tile memory.
wr_addr_iter_id	3	Selects the iterator to add to **wr_addr_base** to compute the address into the selected tile.
wr_addr_iter_vld	1	Validates **wr_addr_iter_id** for computing the address into the selected tile
Note: scratchpad mode controls are only meaningful if **scratchpad_mode==1**		
Snooping controls		
snoop_vld	1	Indicates that data being written to Vbuff should be "snooped" by the max-value-snooping logic on the way into the Vbuff.
snoop_signed	1	if set, use signed-data max-value snooping. If clear, ignore the sign bit and do max-magnitude snooping.
snoop_start_iter_mask	6	When all iterators indicated by the mask are zero, restart the max-value snooping, setting the max snoop value to the current data When the mask is zero, snoop-start is never asserted

14.2 VBUFF-TO-SWITCHBOX SEQUENCER (VB2SB)

This sequencer moves data from the Vbuffer memories to the Switchbox FIFOs. This supports managing the Vbuffer tiles as either a FIFO or a scratchpad. Scratchpad flow control semantics are discussed in section 13.2.

VB2SB Sequencer Op instruction format:

Field	Width	Description
opcd	1	0: Nop 1: RD_VLD. Move data from the Vbuff to a switchbox FIFO.
rd_tile_idx	3	Selects which Vbuff Tile will be read from (managing the Tile as a FIFO)
tgt_select	1	0: write data to AMEM-facing switchbox FIFO 1: write data to Grid-facing switchbox FIFO
scratchpad_mode	1	0: the sequencer reads from the Vbuff in FIFO mode, managing the selected read-tile as a FIFO. 1: the sequencer issues random-access reads, computing the address into the selected tile directly.
Scratchpad Mode controls		
scratchpad_count_iter_mask	6	When all iterators indicated by the mask are zero, **or** if the mask is all-zero, the **scratchpad_count_decr** field is enabed. Otherwise, **scratchpad_count_decr** is disabled and ignored
scratchpad_count_decr	1	if set, **and** if enabled by the **scratchpad_count_iter_mask**, requires the **scratchpad_count** (for the tile selected by **rd_tile_idx**) to be nonzero before the read can be issued, and decrements the count when the read is issued. This provides dependency management for the tile memory, to prevent reading a Vbuffer entry before completion of a pending write. If not set or not enabled by the mask, reads (in scratchpad mode) can proceed without considering the **scratchpad_count**. This may be used when a reading a variable from tile memory when there is no possible dependency on an outstanding write, for example when reading a variable again a 2nd (or later) time.
rd_addr_base	7	Base address for calculating address into the selected tile memory, when using scratchpad mode.
rd_addr_iter_id	3	Selects the iterator, whose value is added to **rd_addr_base** to compute the address into the selected tile.
rd_addr_iter_vld	1	Vallidates **rd_addr_iter_id** for computing the address into the selected tile
Note: scratchpad mode controls are only meaningful if **scratchpad_mode==1**		

14.3 VPU-TO-VBUFF SEQUENCER (VPU2VB)

This sequencer moves data from the VPU OutFIFO to the Vbuffer memories.

Data may be written using FIFO mode semantics, or an address to a tile may be directly computed (in Scratchpad mode) from a base address and an iterator value.

VPU2VB Sequencer Op instruction format:

Field	Width	Description
opcd	1	0: Nop 1: WR_VLD. Move data from the VPU OutFIFO into the Vbuff.
wr_tile_idx	3	Selects which Vbuff Tile will be written to.
scratchpad_mode	1	0: the sequencer writes to the Vbuff in FIFO mode, managing the selected write-tile as a FIFO 1: the sequencer writes to the Vbuff in “scratchpad” mode, which relies on a **scratchpad_count** register to provide flow control
Scratchpad Mode controls		
scratchpad_count_iter_mask	6	When all iterators indicated by the mask have their ending loop value, OR if the mask is all-zero, the **scratchpad_count** register (for the tile indicated by **wr_tile_idx**) is incremented by the **scratchpad_count_incr_amt.**
scratchpad_count_incr_amt	8	Amount to increment the **scratchpad_count** register (if enabled by the **scratchpad_count_iter_mask**) Maximum useful value = 128 (size of one tile)
wr_addr_base	7	Base address for random-access into the selected tile memory.
wr_addr_iter_id	3	Selects the iterator to add to **wr_addr_base** to compute the address into the selected tile.
wr_addr_iter_vld	1	Vallidates **wr_addr_iter_id** for computing the address into the selected tile
Note: scratchpad mode controls are only meaningful if **scratchpad_mode==1**		

14.4 VBUFF-TO-VPU SEQUENCER (VB2VPU)

This sequencer moves data from the Vbuffer memories to the VPU InFIFO. This supports managing the Vbuff tiles as either a FIFO or a scratchpad. Scratchpad flow control semantics are discussed in section 13.2.

VB2VPU Sequencer Op instruction format:

Field	Width	Description
opcd	1	0: Nop 1: RD_VLD. Move data from the Vbuff to VPU InFIFO
rd_tile_idx	3	Selects which Vbuff Tile will be read from.
scratchpad_mode	1	0: the sequencer reads from the Vbuff in FIFO mode, managing the selected read-tile as a FIFO. 1: the sequencer issues random-access reads, computing the address into the selected tile directly.
Scratchpad Mode controls		
scratchpad_count_iter_mask	6	When all iterators indicated by the mask are zero, **or** if the mask is all-zero, the **scratchpad_count_decr** field is enabed. Otherwise, **scratchpad_count_decr** is disabled and ignored.
scratchpad_count_decr	1	if set, **and** if enabled by the **scratchpad_count_iter_mask**, requires the **scratchpad_count** (for the tile selected by **rd_tile_idx**) to be nonzero before the read can be issued, and decrements the count when the read is issued. This provides dependency management for the tile memory, to prevent reading a Vbuffer entry before completion of a pending write. If not set or not enabled by the mask, reads (in scratchpad mode) can proceed without considering the **scratchpad_count**. This may be used when a reading a variable from tile memory when there is no possible dependency on an outstanding write, for example when reading a variable again a 2nd (or later) time.
rd_addr_base	7	Base address for calculating address into the selected tile memory, when using scratchpad mode. Only meaningful if **scratchpad_mode** == 1
rd_addr_iter_id	3	Selects the iterator, whose value is added to **rd_addr_base** to compute the address into the selected tile. Useful for filling VPU registers from tile memory using a loop. Only meaningful if **scratchpad_mode** == 1
rd_addr_iter_vld	1	Vallidates **rd_addr_iter_id** for computing the address into the selected tile Only meaningful if **scratchpad_mode** == 1
Note: scratchpad mode controls are only meaningful if **scratchpad_mode==1**		

aaa