Qualcomm Technologies, Inc.

Qualcomm Hexagon V81 HMX

Programmer's Reference Manual

80-N2040-62 Rev. AA

March 19, 2026

About Qualcomm

Qualcomm relentlessly innovates to deliver intelligent computing everywhere, helping the world tackle some of its most important challenges. Building on our 40 years of technology leadership in creating era-defining breakthroughs, we deliver a broad portfolio of solutions built with our leading-edge AI, high-performance, low-power computing, and unrivaled connectivity. Our Snapdragon® platforms power extraordinary consumer experiences, and our Qualcomm Dragonwing™ products empower businesses and industries to scale to new heights. Together with our ecosystem partners, we enable next-generation digital transformation to enrich lives, improve businesses, and advance societies. At Qualcomm, we are engineering human progress.

Revision history

Revision	Date	Description
AA	March 2026	Initial release

Chapter 1. Introduction

1.1. Purpose

This document describes the Qualcomm Hexagon Matrix eXtension (HMX) instruction set architecture. Hexagon implements HMX alongside an HVX coprocessor. HMX accelerates machine learning workloads by specializing in matrix multiplication and convolution using different data types. It is assumed that the reader is familiar with Hexagon and HVX architecture. For a full description of the Hexagon architecture, see the *Qualcomm Hexagon Programmer's Reference Manual* and *Qualcomm HVX Hexagon Programmer's Reference Manual*.

1.2. Conventions

`Courier font` is used for computer text, registers, and code samples.

For example:

`hexagon_<function_name>_()`.

1.3. Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/ .

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

Chapter 2. HMX features

2.1. Architecture overview of HMX

HMX is a SIMD engine that provides the feature of $y = f(s \cdot \sum_i a_i \cdot w_i + b)$. The CISC-based architecture combines memory accesses with operations.

HMX provides three register sets: a set of accumulator registers, convert state registers, and bias registers.

The accumulator stores the sum of products of the activation and weights input.

$$\text{accumulator} = \sum_i a_i \cdot w_i$$

The convert state stores the result of the accumulator after activation function is applied. Multiple activation functions can apply to convert state, and the result can be stored back to memory.

$$y = f(s \cdot \text{accumulator} + b)$$

The bias register stores the activation function parameters used during the convert operation from accumulator to convert state. The program should load the bias register value from VTCM with a bias load instruction prior to a convert transfer instruction.

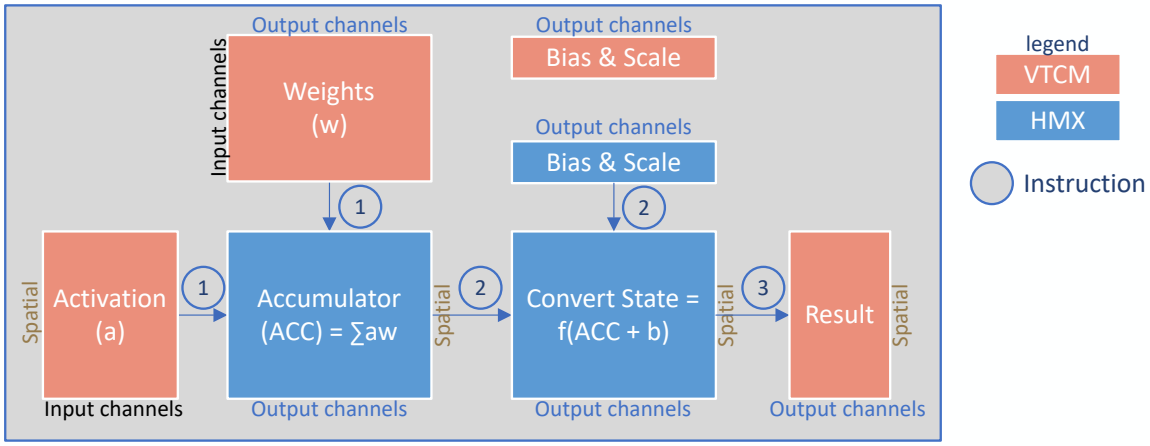

Figure 1. HMX architecture overview

HMX supports three main instructions; the following numbers correspond to the numbered instructions in above figure:

1. Multiply instruction
2. Convert transfer instruction
3. Write convert state instruction

HMX instructions use scalar registers to program the instructions. A basic matrix multiply on HMX looks like the following below where the program loads the activations and weights, executes a convert operation, then stores the data back to memory.

```
    // loads and multiply activation and weights
```

```
    void Q6_activation_hf_mxmem_RR(Word32 Rs, Word32 Rt);
    void Q6_weight_hf_mxmem_RR(Word32 Rs, Word32 Rt);

    // loads bias register from VTCM
    void Q6_bias_mxmem2_A(void* A);

    // convert transfer the accumulator to convert state with bias reg
loaded above
    void Q6_cvt_hf_acc_R(void* A);

    // store result from convert state to vtcm
    void Q6_mxmem_cvt_RR(Word32 Rs, Word32 Rt);
```

NOTE Ensure that only the necessary bits are set when programming HMX, otherwise the behavior is undefined.

NOTE Backwards and forwards compatibility between architecture versions are supported for basic instructions, however, **bit-exactness are not guaranteed.**

2.1.1. SIMD coprocessor

HMX is a single instruction multiple data (SIMD) coprocessor block and the three main instructions all behave in a SIMD manner. The instructions apply their respective equations in parallel to a multitude of data elements at one time.

2.1.2. Data type

HMX instructions support a variety of data types, but this document only covers FP16. HMX FP16 follows IEEE-745 half precision format.

2.2. Multiply instruction

HMX reads in the activation (a_i) and weight (w_i) data that is stored in VTCM, computes the dot-product (multiply accumulate, also known as MAC), and saves the result in an accumulator. The instruction operands specify the memory location and the operation to perform.

The activation and weight instructions must be sequential in the program. Together, they form a multiply instruction. For example:

```
    void Q6_activation_hf_mxmem_RR(Word32 Rs, Word32 Rt);
    void Q6_weight_hf_mxmem_RR(Word32 Rs, Word32 Rt);
```

These instructions performs the following equation:

$$\text{accumulator} = \sum_i a_i w_i \qquad (1)$$

where a = activation, w = weights, i = input.

2.2.1. Activation crouton

An activation crouton is the basic unit in HMX that represents a tensor being operated on by the weights. Each multiply instruction operates on at least one crouton.

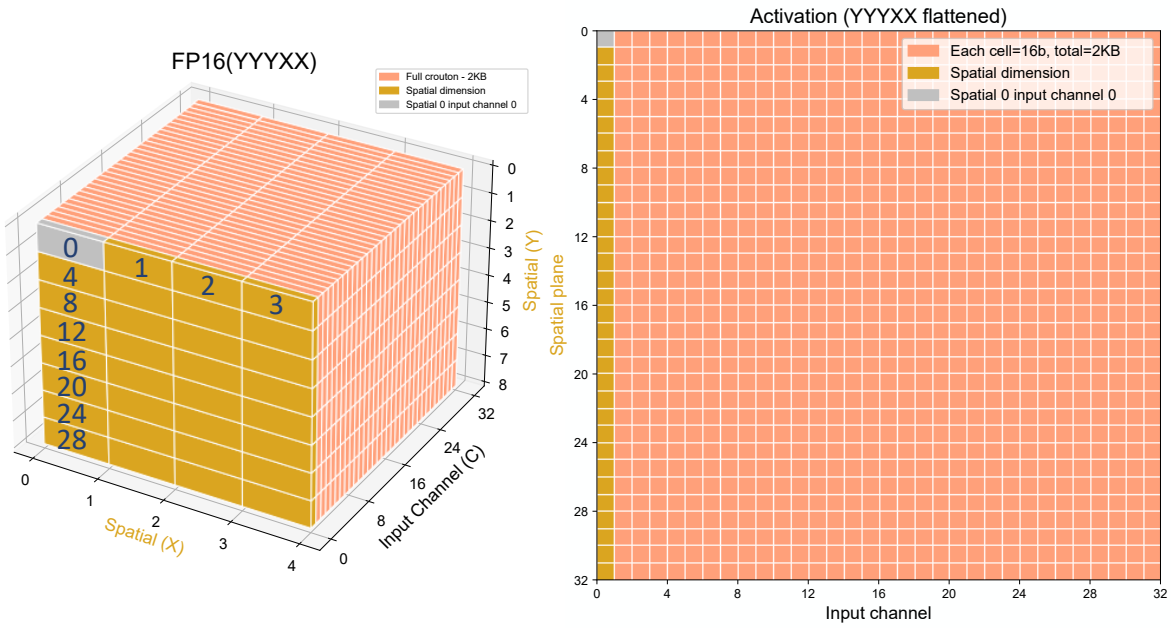

Figure 2. An activation crouton of size 2 KB

A crouton is organized as a 3D structure of cells with dimensions spatial-x by spatial-y by input-channel. Each cell in the crouton is an FP16 value. The spatial-x and spatial-y dimensions together compose the spatial dimension when the activation crouton is viewed as a 2D structure. An FP16 activation crouton is 32 (spatial) x 32 (input-channel) x 16-bit (cell).

Activation instruction format

Activation Instruction Registers					
Oprand	31 11	10 7	6 2	1	0
Rs	Address (2 KB aligned)	Spatial offset (SO, upper 4)	Input channel start	SO	-
Rt	dY (addr offset) or dC (crouton count)	Spatial mask (SM, upper 4)	Input channel stop	SM	-

Figure 3. Activation scalar register fields bit map

Table 1. Activation register fields

Register	Name	Description
Rs32[31:11]	Address	2 KB aligned address of crouton
Rs32[10:7], Rs32[1]	Offset	Spatial offset amount. See Section 4.4.1.
Rs32[6:2]	Input channel start	First input channel to multiply. See Section 4.1.
Rt32[31:11]	dY or dC	Location of the next crouton or number of croutons. See Section 4.2.1, Section 4.4.1.
Rt32[10:7], Rt32[1]	Spatial mask	Organization of the crouton spatial plane
Rt32[6:2]	Input channel stop	Last input channel to multiply. See Section 4.1.

The `Rs[address]` field specifies the activation crouton location in VTCM. It must be 2 KB aligned.

The `Rt[spatial_mask]` defines ordering of the 32 spatials in the spatial-plane using five spatial bits. Set

a spatial mask bit to 0 to indicate the spatial-x (X) direction. Set a spatial mask bit to 1 to indicate the spatial-y (Y) direction. The examples in this document use 4x8 spatial croutons with spatial mask set to 11100 (YYYXX).

The `Rt[dY_dC]`, `Rs[offset]`, `Rs[channel_start]` and `Rt[channel_stop]` fields are used for more complex matrix multiplication, which is covered in Chapter 4. For basic matrix multiplication, set the fields to following values:

- `Rt[dY_dC]=0`
- `Rs[offset]=0`
- `Rs[channel_start]=0`
- `Rt[channel_stop]=31`

These fields must all be defined to perform the specified multiply operation.

Activation VTCM organization

There are 2 activation spatials per VTCM vector as shown in the figure below. The cells colored in blue should be stored in the first 32 bit in VTCM, then the ones in vector 1, vector 2, and lastly the salmon cells. The numbers on the image denote the VTCM address for that specific cell.

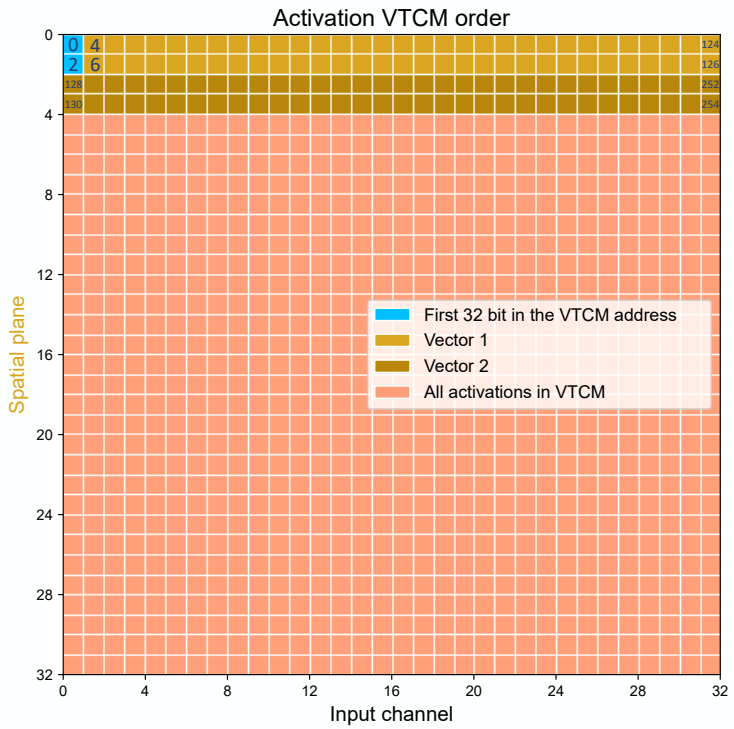

Figure 4. Activation VTCM organization

2.2.2. Weight

A weight instruction applies a set of filters (also known as kernels) to the activation crouton.

Weights are organized as a 2D structure of cells with dimensions of output-channels by input-channels. Each cell in the matrix is an FP16 value. An output-channel is a filter that is applied to the activation data.

Hardware constrains the size of output-channel to 32. The amount of input channels in the weight instruction must be the same as the activation instruction. Each input-channel of the weight correspond to the input channel of the activation crouton.

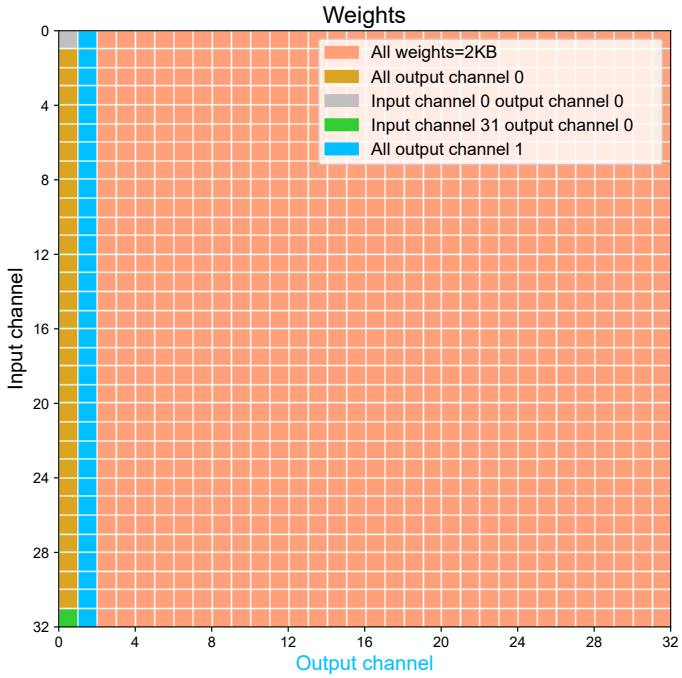

Figure 5. A set of 2KB weights

Weight format

Weight Instruction Registers																
Oprand	31							7	6	5	4					0
Rs	Base address								-	Neg	Reserved (set to all 0s)					
Rt	dW (distance to last weight)								Reserved (set to all 1s to point to last weight)							

Figure 6. Weight scalar register fields bit map

Table 2. Weight instruction register fields

Register	Name	Description
Rs32[31:7]	Address	Starting address of weight data
Rs32[6]	Reserved	Reserved (should be set to 0)
Rs32[5]	Weight negate	Negate the weights such that MAC = A * -W 0: Do not negate 1: Negate
Rs32[4:0]	Reserved	Reserved (should be set to 0)
Rt32[31:7]	dW (range)	Distance to the last weight in the instruction
Rt32[6:0]	Reserved	Reserved (should be set to all 1s)

The `Rt[dW]` field specifies the length of the weight data set, and the amount of the weights must match the activations. The range is only used for exception checking. The sum of `Rs[address]` and `Rt[dW]` should point to the last 128 B vector of the weights data. When too few weights are specified in the multiply and do

not match the activation size, HMX inserts 0 as weights. When too many weights are specified, the hardware wastes cycles flushing these excess weights from the internal buffers.

Weights are read in sequences of 128 B vectors that contain two input channels and 32 output channels as shown in the figure below (bits[6:0]). Subsequent vectors are for the next set of two input channels. For example, the data stored at address = 4 = 0b100 is for output channel 1, input channel 0 (refer to Figure 8).

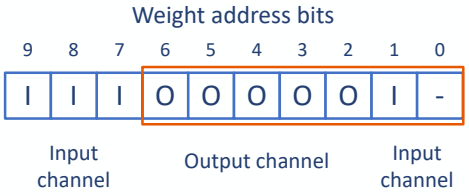

Figure 7. Weights format

HMX reads the input channels in the multiplication instruction order as shown in the figure below. The cells colored in blue should be stored in the first 32 bit in VTCM, then the ones in vector 1, vector 2, and lastly the salmon cells. The numbers on the image denote the VTCM address for that specific cell.

Figure 8. Weights VTCM organization

2.2.3. Accumulator

The accumulator (ACC) stores the MAC result from the multiply instruction.

HMX has a pair of accumulators. Each accumulator is a matrix with a size of spatial dimension by output-channel, which has the size of 32 (spatial) x 32 (output-channel) x 37-bit (cell).

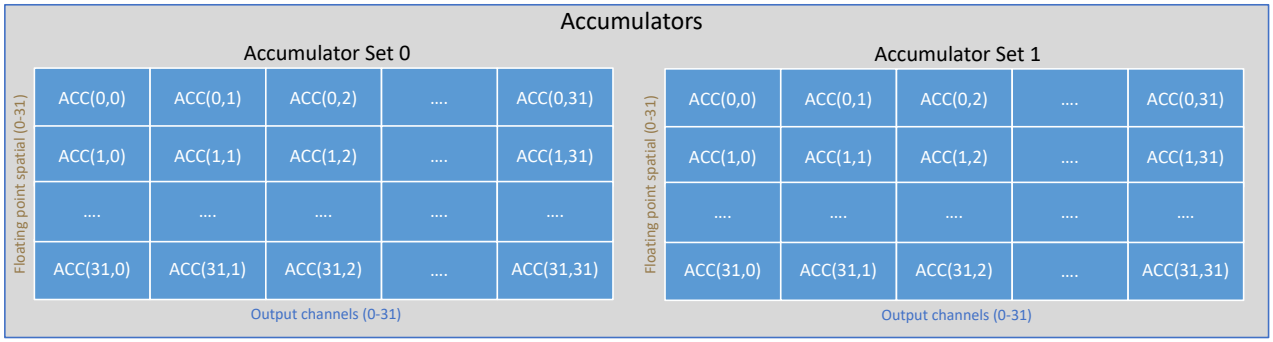

Figure 9. Accumulators

A single matrix multiply instruction executes multiple multiply accumulates (MAC) per cycle.

A matrix multiply operation stores the results in the primary accumulator, and a following convert transfer instruction can optionally clear and swap the accumulator. The convert instructions are the only way to accesses the accumulators and store the data back to VTCM.

2.2.4. Multiply example

The following is an example of matrix multiply using activation crouton, weights, and an accumulator.

From Equation 1 in Section 2.2.1, the cell located at (1,0) in the accumulator stores:

$$ACC(1,\ 0) = \sum_{i} a_{1i} * w_{0i} \qquad (1)$$

where i = input channel, a_1 = activation spatial 1, w_0 = weights output channel 0.

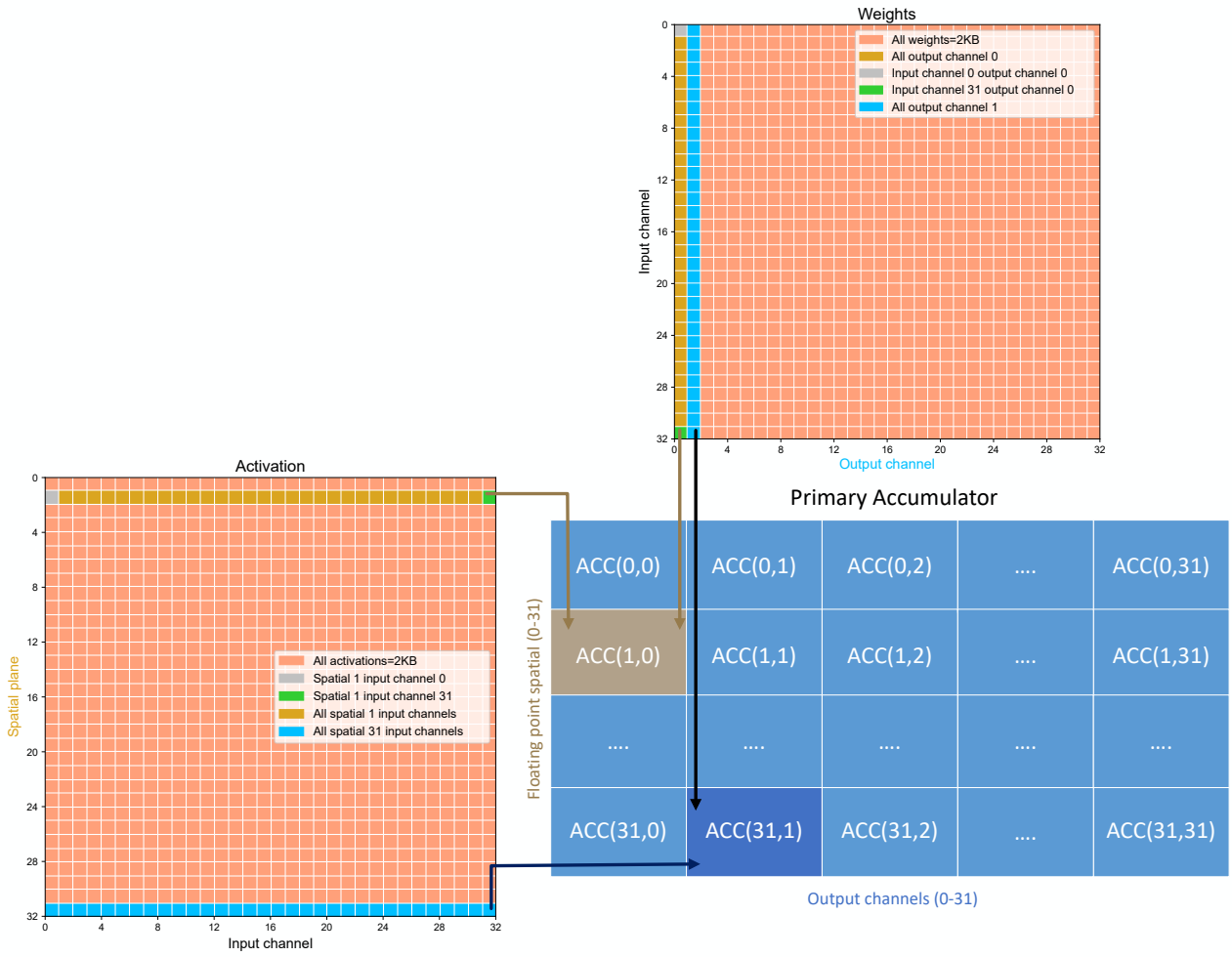

Figure 10. Multiplication result in the primary accumulator

HMX uses the activation data at a spatial dimension and the weight data at an output channel dimension and computes the dot-product across the entire input channel dimension of the data set. In the image above, HMX computes the dot-product starting from the activation and weight gray cell to the green cell. HMX stores the sum of the products from all the gray, gold, and green cells in the accumulator cell at (1, 0). The dot-product across the activation's spatial 1 and weight output-channel 0 determines the accumulator value. This behavior is described by Equation 2 above.

Another example is the blue strips in the activation and weights for activation spatial 31 and weights output-channel 1, with the result stored at accumulator cell (31, 1).

Using Equation 2, the cells in the accumulator as shown above can be expressed as follows:

$$ACC(s, och) = \sum_{i} a_{si} * w_{ochi} \qquad (1)$$

where i = input channel, a_s = activation spatial, w_{och} = weights output channel

A single multiply instruction computes Equation 3 for every activation spatial and weight output-channel in parallel.

2.3. Convert transfer instruction

The convert state transfer instruction uses the bias register values to perform the convert operation on the accumulator data, then transfers the result to the convert state register. Every cell in the accumulator goes

through the same convert operation and is stored in the corresponding location in the convert state register. The instruction then optionally clears the accumulator data. This instruction reduces the 37-bit floating point accumulator value to FP16.

```
void Q6_cvt_hf_acc_R(void* A)
```

This instruction performs the following equation:

$$y = f(s \cdot \text{accumulator} + b) \qquad (1)$$

where b = bias, f = convert operation

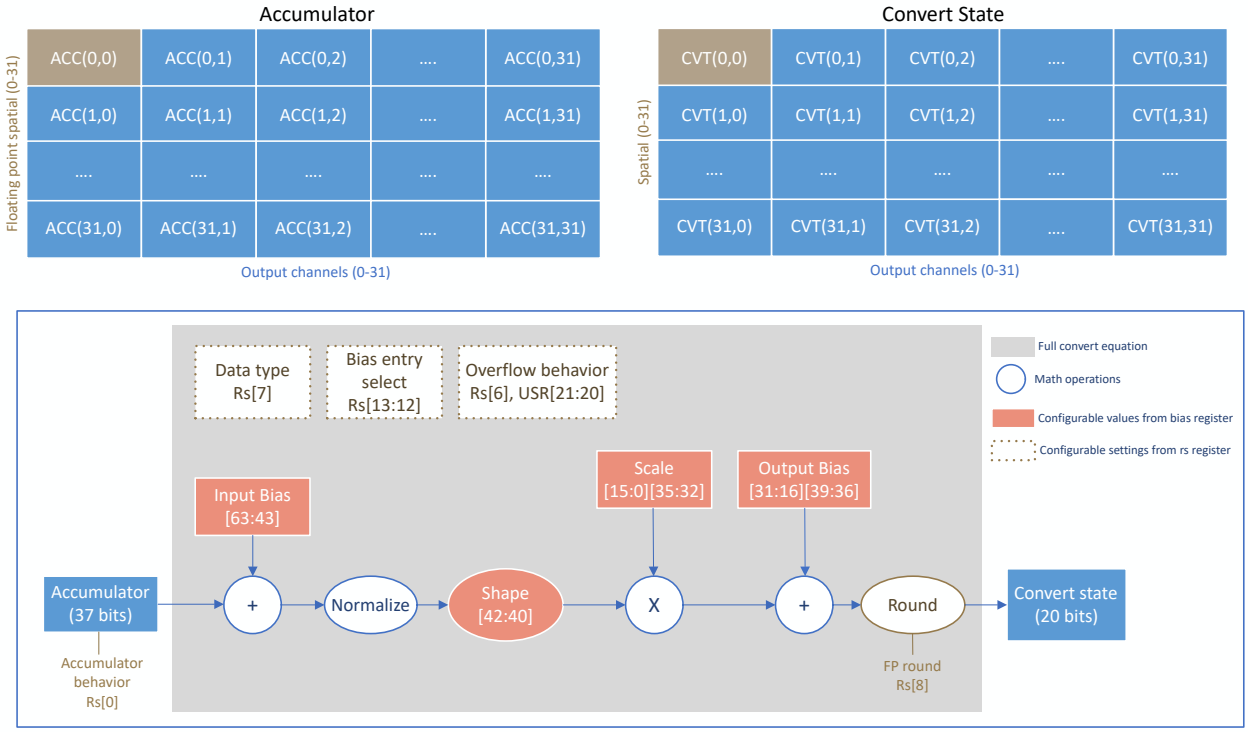

Figure 11. Convert transfer data flow

The gray box encompasses a representation of the convert equation. The boxes with orange background represent the specified bias register parameters. The bias register programs the input bias, shaping function, scale, and output bias. The gold texts with "Rs[bit]" notations represent the modes configured by the Rs register in the convert transfer instruction.

The figure above is expressed as the following equation:

$$\text{convert} = s * \text{shape}(\text{ACC} + b_i) + b_o \qquad (1)$$

Where:

- s = scale
- shape = shaping function
- ACC = accumulator data
- b_i= input bias
- b_o = output bias

2.3.1. Bias register

The bias register specifies the parameters in highlighted in orange in Figure 11.

Each output channel of the accumulator is associated with a bias register (64 bits). A set of bias registers include all bias registers of the 32 output channels. There are four sets of bias registers available to the convert transfer instruction.

Figure 12. Bias register sets

The gold cell in the figure above indicates the bias parameters for output channel 0 used in a convert transfer instruction. A convert transfer instruction selects one of the four sets of bias registers to use, and the bias cells 0 through 31 within the set are applied to the corresponding accumulator output channel values.

Bias load instruction

The bias register must be loaded from VTCM prior to the convert transfer instruction. The instruction loads the data and assigns it to the specified bias register set. A convert transfer instruction uses the preloaded bias data. The bias load instruction can be either before or after the multiplication instructions, as long as it is before the convert transfer instruction.

The bias load instruction is specified as follows:

```
void Q6_bias_mxmem2_A(void* A)
```

This instruction loads 256 bytes of data from VTCM for a bias register set (32 output channels x 64 bits), and the address must be 256 bytes aligned. The data of the 64-bit bias register per output channel is split across two vectors, where the first vector holds the lower 32 bits of all bias registers, and the second vector holds the higher 32 bits of all registers as shown below.

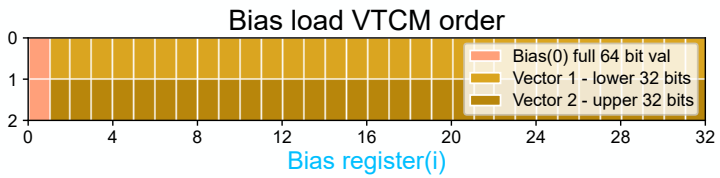

Figure 13. Bias register VTCM organization

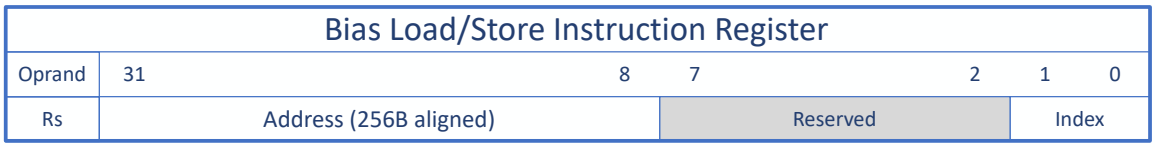

Figure 14. Bias instruction register fields bit map

Table 3. Bias load instruction register fields

Register	Name	Description
Rs32[31:8]	Address	256B aligned address of bias register set.
Rs32[7:2]	Reserved	-
Rs32[1:0]	Index	Bias register set index to load data into.

The 64 bit bias data loaded from VTCM for each output channel is defined as:

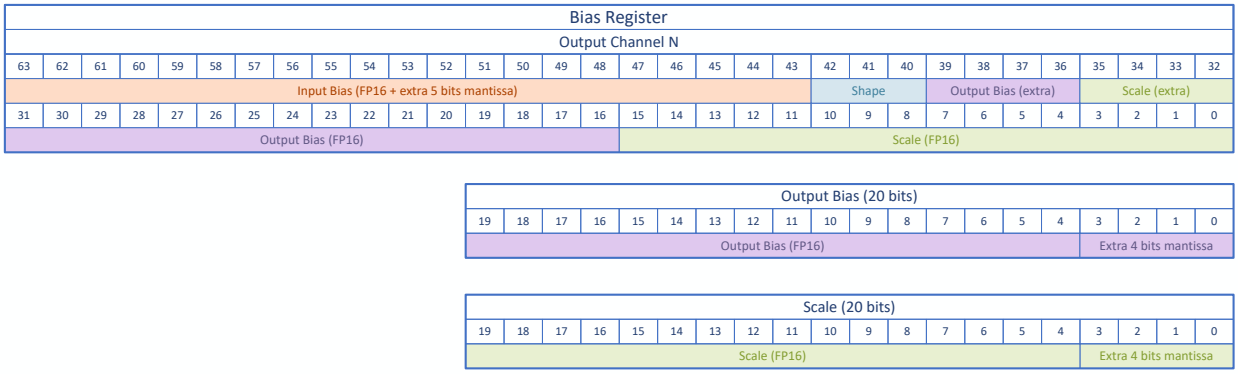

Figure 15. Bias register fields

Table 4. Bias parameters register fields

Register	Name	Description
[63:43]	Input bias	Input bias value (FP16 with 5 extra bits of mantissa)
[42:40]	Shape	Shaping function select
[15:0], [35:32]	Scale	Scale value (FP16 with 4 extra bits of mantissa)
[31:16], [39:36]	Output bias	Output bias value (FP16 with 4 extra bits of mantissa)

The 3-bit shaping field selects the 8 possible shaping functions as listed in the table below:

Table 5. Shaping function options

Shape[2:0]	Shaping functions	Shape[2:0]	Shaping functions
0	shape(x) = x	4	shape(x) = -x
1	shape(x) = min(x,0)	5	shape(x) = -min(x,0)
2	shape(x) = max(x,0)	6	shape(x) = -max(x,0)

Shape[2:0]	Shaping functions	Shape[2:0]	Shaping functions
3	shape(x) = |x|	7	shape(x) = -|x|

Bias store instruction

The bias store instruction stores the bias register used in the convert transfer instruction back to VTCM. It stores 64 bits per output channel and the address must be 256 bytes aligned in VTCM.

The bias store instruction is specified as follows:

```
void Q6_mxmem2_bias_A(void* A)
```

The register field is the same as the bias load instruction above (Figure 14).

2.3.2. Convert state

The convert state stores the result from the convert transfer instruction. It has 32 spatials and 32 output channels. This is similar to the accumulator, as each cell in the accumulator maps to its equivalent location in convert state register.

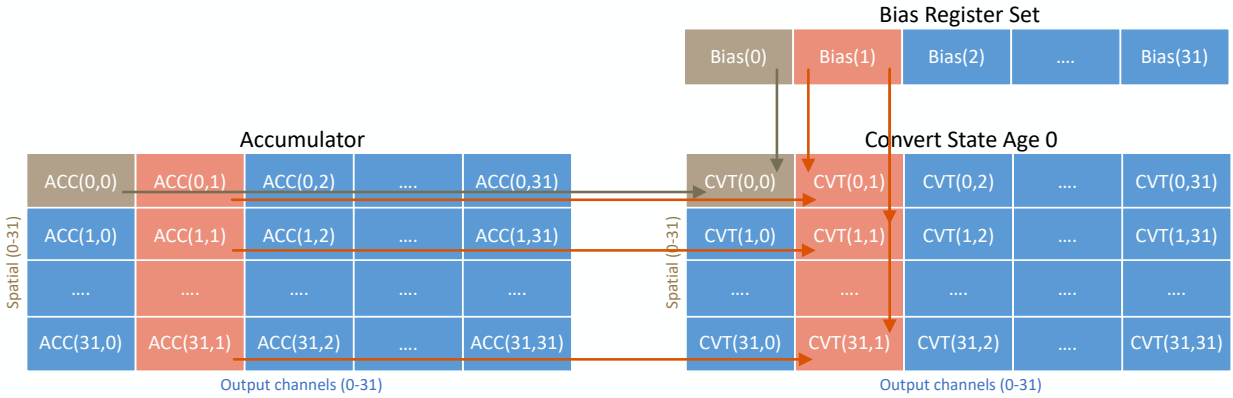

Figure 16. Convert transfer result in a convert state register

In the figure above, accumulator (0,0) is converted with the parameters from bias (0) and stored at convert state (0,0). Each output channel of the accumulator is converted with their own bias parameters. In this figure, bias(0) and bias(1) can contain different parameters for the operation. The cells in accumulator output channel 1 all use the same bias parameters from bias(1).

2.3.3. Convert transfer instruction setup

```
void Q6_cvt_hf_acc_R(void* A)
```

Figure 17. Convert transfer instruction register fields bit map

Table 6. Convert transfer instruction scalar register fields

Register bits	Name	Description
[31:14]	Reserved	-
[13:12]	Bias sel	Select the bias register set for this instruction.
[11:9]	Reserved	-
[8]	Rnd	Rounding position for polynomial feedback. 0: IEEE precision 1: Extra four bits of precision in mantissa
[7]	Reserved	-
[6]	Ovf	Overflow behavior 0: Overflow to +/- infinity 1: Overflow to maxnorm (largest real number)
[4]	Fb op	Selects the limit operation for the selected feedback destination. 0: min 1: max
[3:2]	Fb dest	Specifies the feedback value destination for either output bias or scale. 0: No feedback 1: Output bias 2: Scale
[1]	Reserved	-
[0]	acc	Accumulator behavior 0: Clear accumulator values and swap primary accumulator 1: Retain accumulator values

The round position, feedback limit operation and destination fields are used for convert transfer operations with feedback, which is covered in Chapter 5. For convert transfer instruction without feedback, set the fields to following values:

- `Rs[bias_sel]` = the bias register index (the index `Rs[1:0]` field in the bias load instruction prior to this convert transfer instruction)
- Set the rest of the register to 0

2.3.4. Floating point overflow setting

Together, the `USR[21:20]` and `Rs[6]` bits control the behavior of the floating point overflow result. The table below demonstrates the behavior:

Table 7. Floating point overflow control

Control bits			Saturation data		FP16 output		
USR[20] infNan propagate	Rs[6] maxnorm	USR[21] nan propagate	Input	Output	Sign	Exp	Sig
0	x	x	+Inf	+maxpos	0	emax	all 1's
			-Inf	-maxpos	1	emax	all 1's
			Nan	-maxpos	1	emax	all 1's

Control bits			Saturation data		FP16 output		
1	0	x	+Inf	+Inf	0	emax	all 0’s
			-Inf	-Inf	1	emax	all 0’s
			Nan	Nan	1	emax	all 1’s
1	1	0	+Inf	+maxpos	0	emax-1	all 1’s
			-Inf	-maxpos	1	emax-1	all 1’s
			Nan	-maxpos	1	emax-1	all 1’s
1	1	1	+Inf	+maxpos	0	emax-1	all 1’s
			-Inf	-maxpos	1	emax-1	all 1’s
			Nan	-maxpos	1	emax	all 1’s

2.4. Write convert state instruction

The write convert state instruction stores the FP16 data in convert state age to VTCM at the specified address. It does not modify the data in the convert state. The 2D structure of the convert state of dimensions spatial by output channel are stored in VTCM in a 3D structure of the dimensions spatial-x by spatial-y by output channels. The extra dimension is derived from separating the spatial dimension in the convert state into two dimensions of spatial-x and spatial-y.

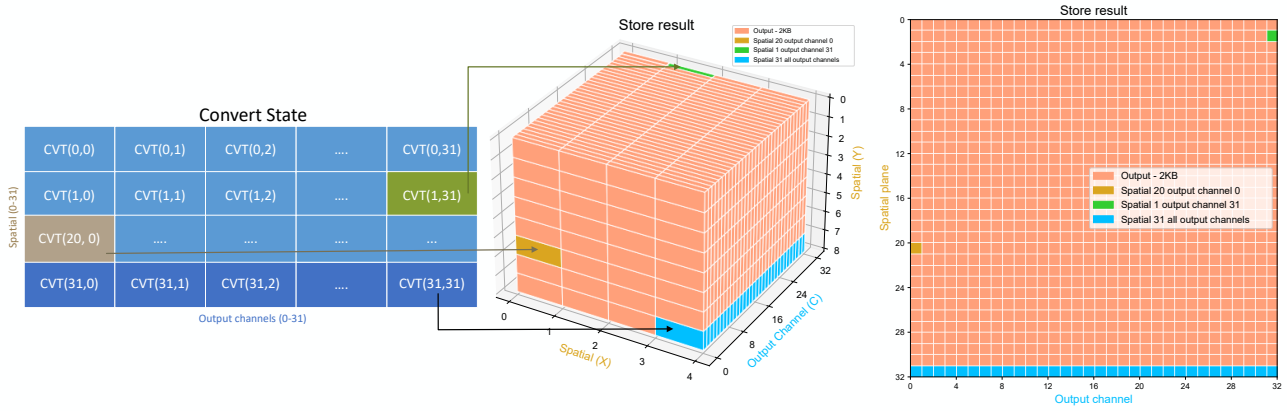

Figure 18. Write convert state instruction spatial mapping

The figure above shows the mapping from convert state to VTCM in both "crouton view" and 2D VTCM view. The instruction stores the entire row of spatial 31 (deep blue) in the convert state to VTCM in the whole depth behind spatial 31. The gold cell in CVT (20, 0) maps to the gold cell in VTCM at the front, and the green cell in CVT (1, 31) maps to the green cell at the last output channel depth in VTCM. The same process is repeated for the full convert state register in this instruction.

2.4.1. Write convert state instruction format

```
void Q6_mxmem_cvt_RR(Word32 Rs, Word32 Rt);
```

Write convert state instruction register fields bit map

Write Convert State Instruction Register					
Oprand	31 11	10 7	6 2	1	0
Rs	Address (2 KB aligned)	Reserved			
Rt	Reserved	Spatial mask (SM, upper 4)	Reserved	SM	-

Table 8. Write convert state instruction register fields

Register	Name	Description
Rs32[31:11]	Address	VTCM address to write to.
Rs32[10:0]	Reserved	-
Rt32[31:11]	Reserved	-
Rt32[10:7], Rt32[1]	Spatial mask	Organization of the crouton spatial plane.
Rt32[6:2], Rt32[0]	Reserved	-

The `Rs[address]` field specifies the activation crouton location in VTCM. It must be 2 KB aligned.

The `Rt[spatial_mask]` defines ordering of the 32 spatials in the spatial-plane using five spatial bits, the same as activation spatial mask Section 2.2.1.1.

VTCM organization

The result is organized similarly to the activation crouton in VTCM with two spatials stored in a vector as shown below.

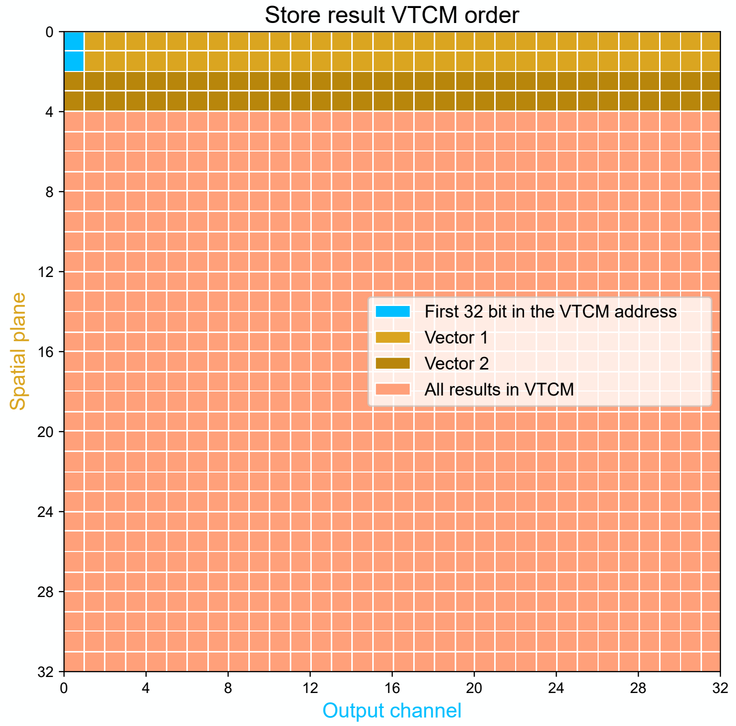

Figure 19. Result VTCM organization

2.5. Clear accumulator instruction

This instruction clears both accumulators in a set by setting the values to zero. The accumulators can also optionally be cleared by the convert transfer instruction.

```
    void Q6_mxclracc_hf()
```

Chapter 3. HMX matrix multiply code example

3.1. Data setup in VTCM

FP16 data in HMX follows IEEE-745 half precision format. This example uses simple FP16 data to demonstrate HMX matrix multiplication and it's not the most efficient. Full application should use HVX to manipulate the data.

```
/*
    Setup the values as:
        0 0.01 0.02 0.03  ....... 0.31
        1 1.01 1.02 1.03  ....... 1.31
        ......
        ......
        31 31.01 ................ 31.31
    spatial = int val, channel = frac val
    rounded to the closest representable FP16 value
*/
void write_act_vtcm(uint16_t *addr_start) {
    int total_sp = 32;
    int total_ch = 32;
    int num_sp_per_vec = 2;   // 2 spatial per vector
    int total_vectors = total_sp / num_sp_per_vec;

    uint16_t* addr = addr_start;
    for (int vec = 0; vec < total_vectors; vec++) {
        for (int ch = 0; ch < total_ch; ch++) {
            for (int sp_in_vec = 0; sp_in_vec < num_sp_per_vec;
sp_in_vec++) {
                int spatial = vec * num_sp_per_vec + sp_in_vec;
                float act_val = spatial + (ch * 0.01);
                uint16_t fp16_val = float_to_fp16(act_val);
                *addr = fp16_val;

                addr++;  // move to next FP16 addr
            }
        }
    }
}


/*
    Setup the values as:
        0 0 0 0 ....... 0
        0 1 0 0 ....... 0
        0 0 2 0 ....... 0
        .....
        .....
        0 0 0 0 ....... 31

    essentially "identity" matrix but with none 0s location as the
output channel number
*/
void write_wgt_vtcm(uint16_t *addr_start) {
    int total_out = 32;
    int total_in = 32;
    int num_in_per_vec = 2;   // 2 spatial per vector
    int total_vectors = total_in / num_in_per_vec;

    uint16_t* addr = addr_start;
    for (int vec = 0; vec < total_vectors; vec++ ) {
```

```
        for (int out = 0; out < total_out; out ++ ) {
            for (int in_ch_vec = 0; in_ch_vec < num_in_per_vec;
in_ch_vec++) {
                int input_channel = vec * num_in_per_vec + in_ch_vec;
                float wgt_val = 0;
                if (input_channel == out) {
                    wgt_val = out;
                }
                uint16_t fp16_val = float_to_fp16(wgt_val);
                *addr = fp16_val;

                addr++;  // move to next FP16 addr
            }
        }
    }
}

/*
    Setup the values to be the same for every output channel:
        * input bias = 0
        * shape = 0 (x)
        * scale = 1
        * output bias = 0
    So that the converted value is the "raw" ACC value
*/
void write_bias_vtcm(int32_t *addr_start) {
    int total_bias = 32;

    int32_t* addr = addr_start;
    // lower 32 bits of all output channel bias regs
    for (int out = 0; out < total_bias; out++) {    // every output
channel
        float scale_val = 1;
        uint16_t fp16_val = float_to_fp16(scale_val);
        *addr = fp16_val;   // only scale is set at [15:0]. No need for
extra bits here

        addr++;  // move to next 32 bit bias addr
    }

    // upper 32 bits of all output channel bias regs (all 0s here)
}
```

Bias stored in vtcm in the above code are the same for every output channels for illustration purposes. Each output channel does not have to be the same.

3.2. Code

```
    // find VTCM address
    void *vtcm_addr = setup_vtcm();

    // ------------------------------------
    // --- set act/wgt/bias data in VTCM ---
    // ------------------------------------
    uint8_t *act_start_addr = (uint8_t*) vtcm_addr;
    write_act_vtcm((uint16_t*)act_start_addr);
    int crouton_size = 2048;

    int8_t *wgt_start_addr = (int8_t*) (act_start_addr + crouton_size);
    write_wgt_vtcm((uint16_t*)wgt_start_addr);
    int wgt_size = 2048;
```

```
    int32_t *bias_start_addr = (int32_t*)(wgt_start_addr + wgt_size);
    write_bias_vtcm(bias_start_addr);
    int bias_size = 256;

    uint8_t *result_start_addr = (uint8_t *) vtcm_addr + 3 * 2048;  //
next 2 kB aligned addr

    // -------------------------------------
    // --- setup multiply regs and insn ----
    // -------------------------------------
    uintptr_t act_rs = (uintptr_t)act_start_addr;  // other non addr
params = 0

    uint32_t act_rt = 0;  // dY_dC = 0
    const uint32_t rt_sp_mask = 0b11100;    // YYYXX
    const uint32_t rt_in_stop = 31;
    act_rt |= ( ((rt_sp_mask >> 1) << 7 ) | (rt_in_stop << 2 ) |
((rt_sp_mask & 0x1) << 1) );

    uintptr_t wgt_rs = (uintptr_t)wgt_start_addr;
    uint32_t wgt_rt = wgt_size - 1;

    Q6_activation_hf_mxmem_RR(act_rs, act_rt);
    Q6_weight_hf_mxmem_RR(wgt_rs, wgt_rt);

    // -------------------------------------
    // --- setup convert regs and insn -----
    // -------------------------------------

    // load bias first
    int32_t *bias_rs = bias_start_addr; // loading to bias index 0
    Q6_bias_mxmem2_A(bias_rs);

    // all convert params are 0, including using bias index 0 from
earlier
    uint32_t cvt_rs = 0;
    Q6_cvt_hf_acc_R(cvt_rs);

    // -------------------------------------
    // ----- setup write regs and insn -----
    // -------------------------------------
    uint8_t* wr_rs = result_start_addr;
    uint32_t wr_rt = ( ((rt_sp_mask >> 1) << 7 ) | ((rt_sp_mask & 0x1)
<< 1) ); // same spatial mask as act
    Q6_mxmem_cvt_RR(wr_rs, wr_rt);
```

Chapter 4. HMX multiply variants

4.1. Multiply less input channels

HMX supports specifying a input channel start and input channel stop in the instruction to limit the number of input channels to less than the full 32 channels (Figure 10) in a standard size matrix multiplication. This decreases the computation for all 32 weight filters.

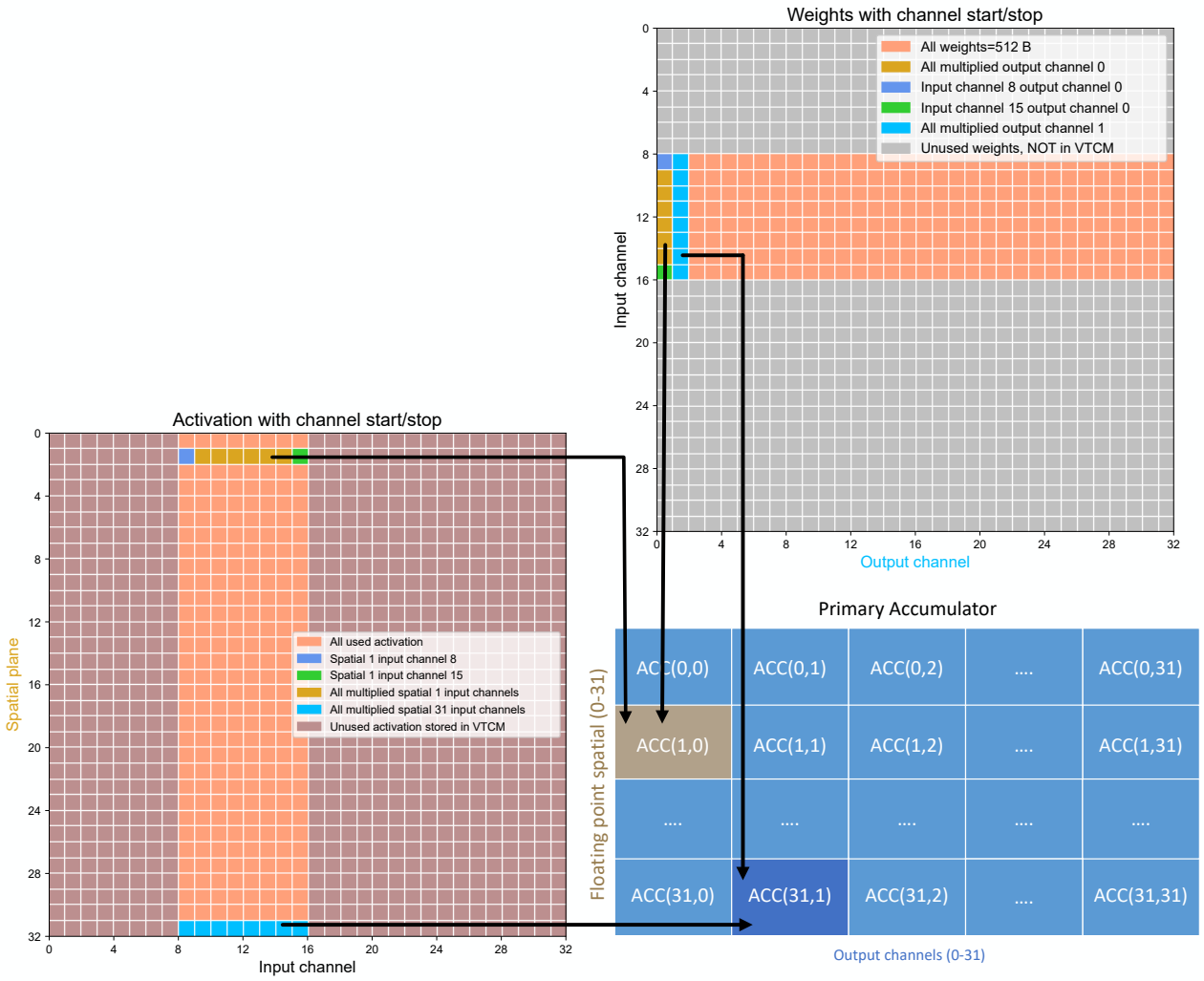

Figure 20. Example multiplication of input channel range 8-15 in the primary accumulator

4.1.1. Instruction setup

Set the `Rs[channel_start]` and `Rt[channel_stop]` fields in the activation instruction as seen in Table 1. The start input channel must be smaller than the stop input channel, and the starting channel must be increments of 8. The full channel range also must be in increments of 8. i.e the example above sets `Rs[channel_start]=8` and `Rt[channel_stop] = 15`.

NOTE

The full 2 KB activation crouton must be stored in VTCM regardless of the channel range setting. However, only the used weight channels should be stored in VTCM. i.e. the weight address should point to the first weight used (input channel 8 in above example), and the weight range should be the total weight needed (4 vectors).

4.1.2. Code example

Using the same VTCM setup as Section 3.1, replace the multiplication instruction register setup section in the code in Section 3.2 with the below code:

```
    uintptr_t act_rs = (uintptr_t)act_start_addr;
    const uint32_t rs_in_start = 8;  // start channel 8 aligned
    act_rs |=  (rs_in_start << 2 );

    uint32_t act_rt = 0;  // dY_dC = 0
    const uint32_t rt_sp_mask = 0b11100;    // YYYXX
    const uint32_t rt_in_stop = 15;  // total channels 8 aligned
    act_rt |= ( ((rt_sp_mask >> 1) << 7 ) | (rt_in_stop << 2 ) |
((rt_sp_mask & 0x1) << 1) );

    const int vec_size = 128;
    uintptr_t wgt_rs = (uintptr_t)wgt_start_addr + vec_size *
rs_in_start / 2; // didn't change vtcm fill, so point to start ch
address
    // usually it should just fill less data in VTCM
    uint32_t less_wgt_size = vec_size * (rt_in_stop + 1 - rs_in_start) /
2;
    uint32_t wgt_rt = less_wgt_size - 1;

    Q6_activation_hf_mxmem_RR(act_rs, act_rt);
    Q6_weight_hf_mxmem_RR(wgt_rs, wgt_rt);
```

4.2. Multiply more input channels (multiple croutons)

HMX supports multiplying multiple croutons in an instruction, up to 32 croutons (1024 input channels).

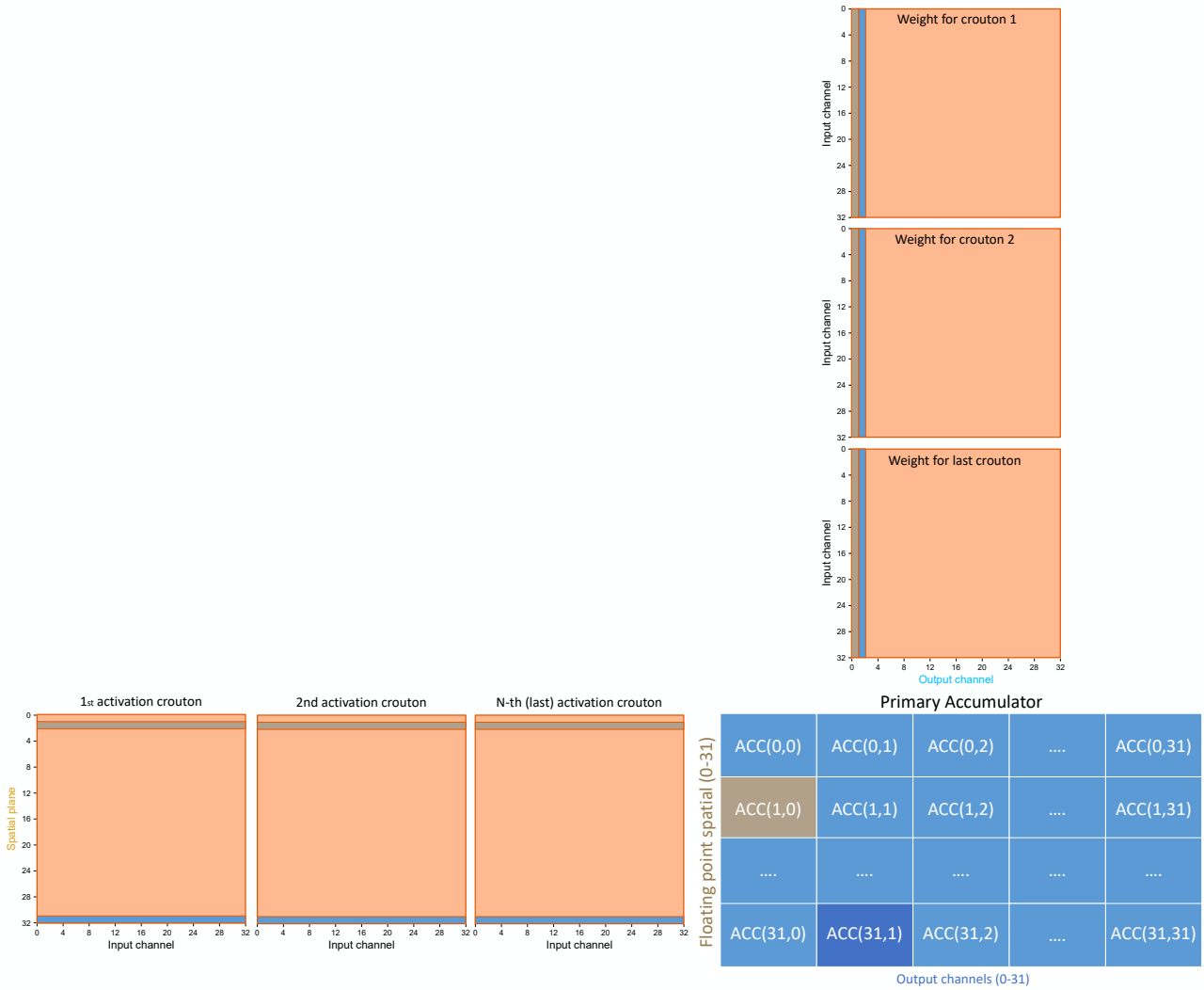

Figure 21. Example multiplication of n croutons

4.2.1. Instruction setup

Use the "activation deep" instruction for this multiplication. The weight instruction paired with this activation instruction is the same, however, the weight range field (`Rt[dW]`) should be updated to reflect the bigger range in this multiplication.

```
void Q6_activation_hf_mxmem_RR_deep(Word32 Rs, Word32 Rt);
void Q6_weight_hf_mxmem_RR(Word32 Rs, Word32 Rt);
```

The activation deep instruction has the same register fields as a regular activation instruction as seen in Table 1.

In the activation deep instruction, the `Rt[dY_dC]` field is treated as `Rt[dC]` where the value + 1 denotes the number of croutons in this instruction. Setting `Rt[dC]=0` (1 crouton) is the same as using a regular activation instruction. Maximum value supported is `Rt[dC]=31` (32 croutons).

NOTE The activation croutons must be continuous in VTCM. The weights also must be continuous in VTCM.

4.2.2. Code example

Below code is an example multiplying 3 croutons. It uses similar VTCM setup as Section 3.1, but there must be 3 consecutive `write_act_vtcm` to fill 3 continuous croutons. It's then followed by 3 consecutive `write_wgt_vtcm` to write the weights for the corresponding croutons. The bias values in vtcm can stay the same.

Replace the multiplication instruction register setup section in the code in Section 3.2 with the below code:

```
    uintptr_t act_rs = (uintptr_t)act_start_addr;  // other non addr
params = 0

    int num_croutons = 3;
    uint32_t act_rt = ((num_croutons-1) << 11);  // dC + 1 = total
croutons
    const uint32_t rt_sp_mask = 0b11100;    // YYYXX
    const uint32_t rt_in_stop = 31;
    act_rt |= ( ((rt_sp_mask >> 1) << 7 ) | (rt_in_stop << 2 ) |
((rt_sp_mask & 0x1) << 1) );

    uintptr_t wgt_rs = (uintptr_t)wgt_start_addr;
    uint32_t wgt_size = 2048 * num_croutons;
    uint32_t wgt_rt = wgt_size - 1;

    Q6_activation_hf_mxmem_RR_deep(act_rs, act_rt);  // deep instruction
    Q6_weight_hf_mxmem_RR(wgt_rs, wgt_rt);
```

4.2.3. Partially multiply multiple croutons

HMX supports specifying a input channel start and input channel stop in the instruction to limit the number of input channels in deep activation instruction as well, similar to Section 4.1. However, the start and end only applies to the first and last crouton respectively.

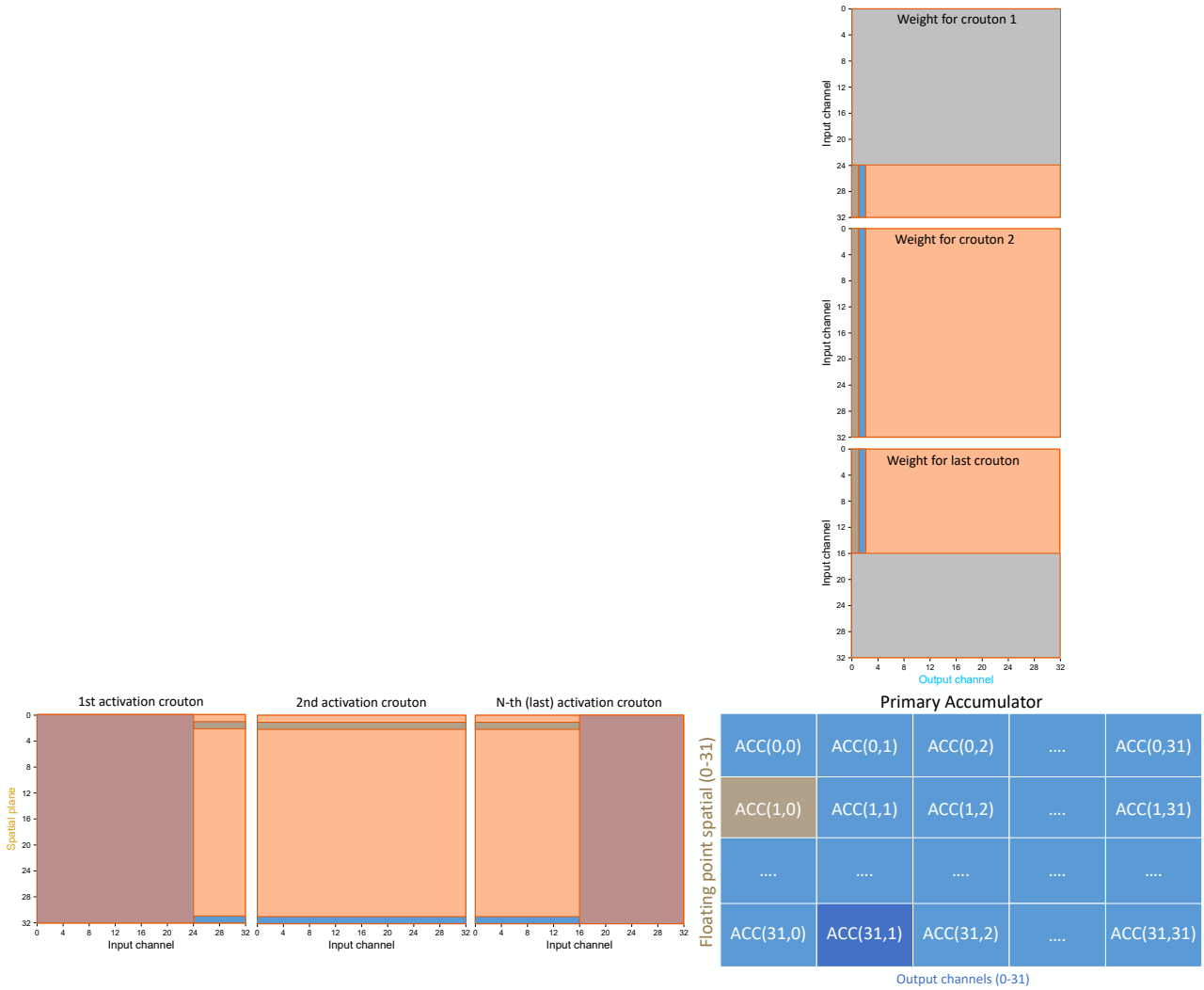

Figure 22. Example multiplication of n croutons with channel start and stop

The example above sets `Rs[channel_start]=24` and `Rt[channel_stop]=15` while using the activation deep instruction.

4.2.4. Instruction setup

Set the `Rs[channel_start]` and `Rt[channel_stop]` fields in the activation instruction as seen in Table 1.

- `Rs[channel_start]` field applies to the first crouton
- All 32 input channels in the middle croutons are computed.
- `Rt[channel_stop]` applies to the last crouton.

Since the start input channel and the stop input channel fields apply to different croutons, `Rs[channel_start]` can be greater than `Rt[channel_stop]`. The limitation of the the starting channel and full channel range must be in increments of 8 still applies.

4.2.5. Code example

Below code is an example multiplying 3 croutons. Using the same VTCM setup as Section 4.2.2, replace the multiplication instruction register setup section in the code in Section 3.2 with the below code:

```
    uintptr_t act_rs = (uintptr_t)act_start_addr;
```

```
    const uint32_t rs_in_start = 24;  // start channel 8 aligned
    act_rs |=  (rs_in_start << 2 );

    uint32_t act_rt = ((num_croutons-1) << 11);  // dC + 1 = total
croutons
    const uint32_t rt_sp_mask = 0b11100;     // YYYXX
    const uint32_t rt_in_stop = 15;  // total channels 8 aligned. stop
can < start for this
    act_rt |= ( ((rt_sp_mask >> 1) << 7 ) | (rt_in_stop << 2 ) |
((rt_sp_mask & 0x1) << 1) );

    const int vec_size = 128;
    uintptr_t wgt_rs = (uintptr_t)wgt_start_addr + vec_size *
rs_in_start / 2; // didn't change vtcm fill, so point to start ch
address
    // usually it should just fill less data in VTCM

    uint32_t less_wgt_size = vec_size * ((32-rs_in_start) +
(num_croutons-2)*32 + (rt_in_stop + 1)) / 2;
    uint32_t wgt_rt = less_wgt_size - 1;

    Q6_activation_hf_mxmem_RR_deep(act_rs, act_rt);  // deep instruction
    Q6_weight_hf_mxmem_RR(wgt_rs, wgt_rt);
```

4.3. Multiply more filters (64 filters)

HMX supports multiplying up to 64 filters in a pair of multiply instructions with "weight deep" instruction. This instruction is functionally the same as two back-to-back multiply instructions on the same activation crouton with different weights.

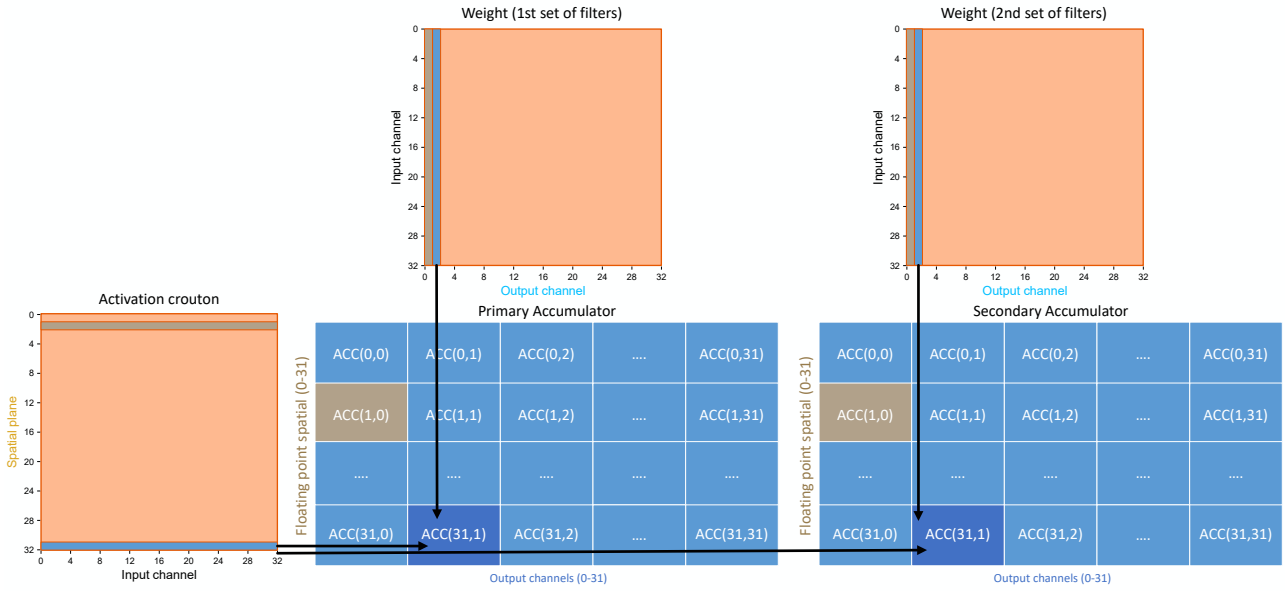

Figure 23. Example multiplication of 64 filters

4.3.1. Instruction setup

The weight deep instruction has the same register fields as a regular weight instruction as seen in Table 2. The weight range field (`Rt[dW]`) should be updated to reflect the bigger range in this multiplication.

```
    void Q6_activation_hf_mxmem_RR(Word32 Rs, Word32 Rt);
    void Q6_weight_hf_mxmem_RR_deep(Word32 Rs, Word32 Rt);
```

This instruction stores the result in both accumulators, so it needs 2 sets of convert transfer and convert store instructions to fully write out the result to VTCM.

4.3.2. Code example

Use similar VTCM setup as Section 3.1, but there should be 2 consecutive `write_wgt_vtcm` to write the data for the additional 32 filters. The activation and bias data can stay the same.

Replace the multiplication instruction register setup section in the code in Section 3.2 with the below code:

```
    uintptr_t act_rs = (uintptr_t)act_start_addr;  // other non addr
params = 0

    uint32_t act_rt = 0;  // dY_dC = 0
    const uint32_t rt_sp_mask = 0b11100;    // YYYXX
    const uint32_t rt_in_stop = 31;
    act_rt |= ( ((rt_sp_mask >> 1) << 7 ) | (rt_in_stop << 2 ) |
((rt_sp_mask & 0x1) << 1) );

    uintptr_t wgt_rs = (uintptr_t)wgt_start_addr;
    uint32_t wgt_size = 2048 * 2;
    uint32_t wgt_rt = wgt_size - 1;

    Q6_activation_hf_mxmem_RR(act_rs, act_rt);
    Q6_weight_hf_mxmem_RR_deep(wgt_rs, wgt_rt);  // deep instruction
```

Additionally, there should be two sets of convert transfer instructions followed by convert write instructions to write out the full 64 output channels to vtcm. The full instruction sequence should be:

1. Multiply with weight deep
2. Load bias for first convert transfer instruction
3. First convert transfer instruction (for filters 0-31)
4. First convert write instruction (for filters 0-31)
5. Load bias for second convert transfer instruction
6. Second convert transfer instruction (for filters 32-63)
7. Second convert write instruction (for filters 32-63)

4.4. Multiply unaligned croutons

HMX supports multiplying unaligned croutons in the Y direction by using data from two croutons.

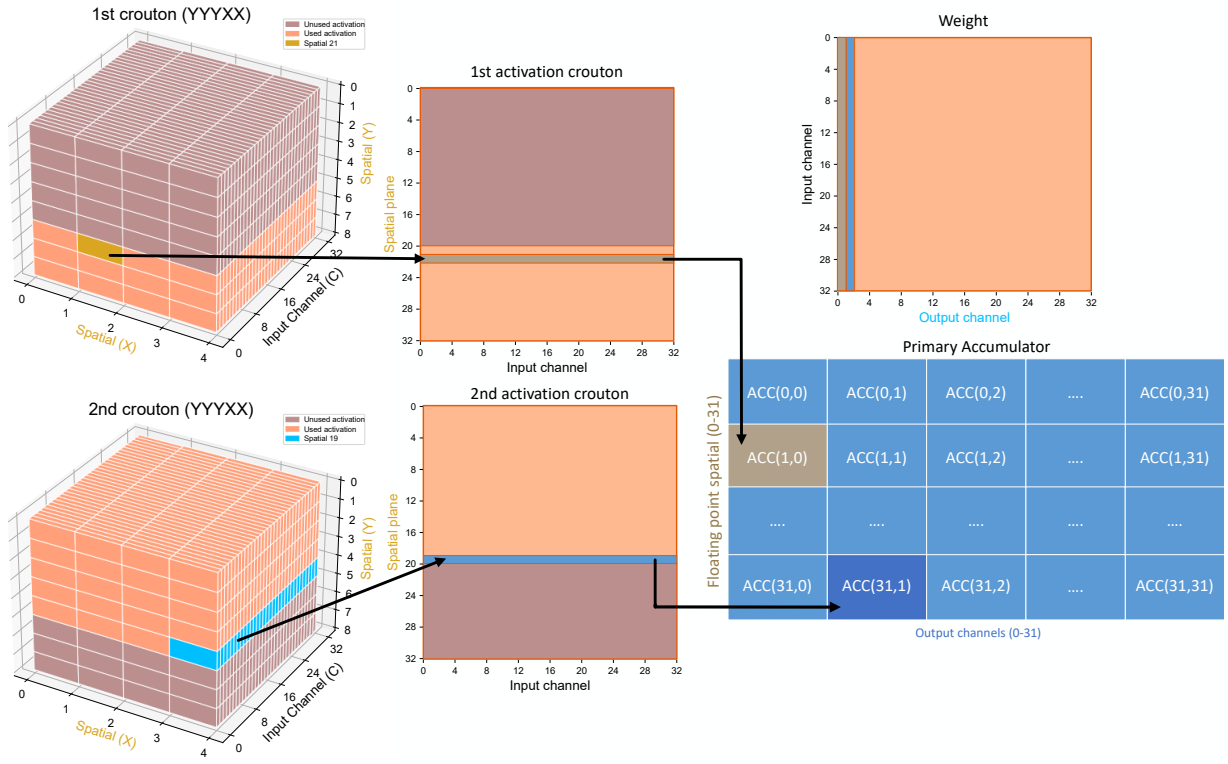

Figure 24. Example multiplication with y=5 activation single instruction

The figure above shows that with y=5 offset, multiplication only starts at spatial 20. The result of the 1st crouton's spatial 21 is stored in accumulator spatial 1. The first 5 rows of the 2nd crouton are also multiplied in this instruction, with the result of spatial 19 of the 2nd crouton stored in accumulator spatial 31.

4.4.1. Instruction setup

Use the "activation single" instruction for this multiplication. The weight instruction paired with this activation instruction remains the same as a regular matrix multiply. The activation single instruction will take the bottom portion of the first crouton and the top portion of the second crouton to form a full 2KB crouton in matrix multiplication.

```
void Q6_activation_hf_mxmem_RR_single(Word32 Rs, Word32 Rt);
void Q6_weight_hf_mxmem_RR(Word32 Rs, Word32 Rt);
```

The activation single instruction has the same register fields as a regular activation instruction as seen in Table 1.

In this instruction, the `Rt[dY_dC]` field is treated as `Rt[dY]` where the value is the offset to the 2nd crouton. `Rs[address]` + `Rt[dY]` = the starting address of the 2nd crouton in VTCM. `Rt[dY]` can be a negative value.

The `Rs[offset]` field specifies the amount of unaligned offset in the first crouton. The field's X and Y bits must match the `Rt[spatial_mask]` field. The `Rt[spatial_mask]` applied to `Rs[offset]` field determines the amount of offset in the x or y direction. For activation single instruction, set the x bits corresponding `Rs[offset]` field to 0, and the y bits to the amount of offsets needed. i.e. in the example above, `Rt[spatial_mask] = 11100 = YYYXX`, and it has offset of 5 in the Y direction by setting `Rs[offset] = 10100` (last two bits are x direction = 0b00 = 0, top 3 bits are for y direction = 0b101 = 5).

The `Rs[channel_start]` and `Rt[channel_stop]` fields are also supported in this instruction and

behave the same on the input channel dimension as a regular activation instruction (Section 4.1).

4.4.2. Code example

Use similar VTCM setup as Section 3.1, but there should be 2 `write_act_vtcm` total to write the data for both croutons (this example code uses consecutive croutons). The weight and bias data can stay the same.

Replace the multiplication instruction register setup section in the code in Section 3.2 with the below code:

```
    uintptr_t act_rs = (uintptr_t)act_start_addr;
    const uint32_t rs_offset = 0b10100;  // Y=5, X=0
    act_rs |= ( ((rs_offset >> 1) << 7 ) | ((rs_offset & 0x1) << 1) );
    int crouton_size = 2048;

    uint32_t act_rt = crouton_size;  // dY = 1 crouton away from base
address
    const uint32_t rt_sp_mask = 0b11100;    // YYYXX
    const uint32_t rt_in_stop = 31;
    act_rt |= ( ((rt_sp_mask >> 1) << 7 ) | (rt_in_stop << 2 ) |
((rt_sp_mask & 0x1) << 1) );

    uintptr_t wgt_rs = (uintptr_t)wgt_start_addr;
    uint32_t wgt_size = 2048;
    uint32_t wgt_rt = wgt_size - 1;

    Q6_activation_hf_mxmem_RR_single(act_rs, act_rt);   // single
instruction
    Q6_weight_hf_mxmem_RR(wgt_rs, wgt_rt);
```

Chapter 5. HMX convert transfer with feedback

The convert state operation can optionally use feedback to support polynomial activation functions. The operation is similar to a normal convert operation but uses the result of a previous convert transfer instruction as feedback for the scale or output bias of the operation.

The convert transfer instruction scalar register configures the following feedback options for convert state operation:

- Select a feedback value to represent as the output bias or scale
- Use a min or max function to compare the feedback value with the bias register value

These options allow different convert equation possibilities.

Table 9. Convert feedback bias and scale equation options

Equation number	Equation
1	$C = sX + \min(b, C')$
2	$C = sX + \max(b, C')$
3	$C = \min(s, C')X + b)$
4	$C = \max(s, C')X + b)$

Where

- C = The current convert state being computed
- C' = The previous convert state used for feedback
- $X = text\{shape\}((text\{AC\} + b_i)$ (this equation is the partial equation of Equation 5, also colored in light blue in figure below)

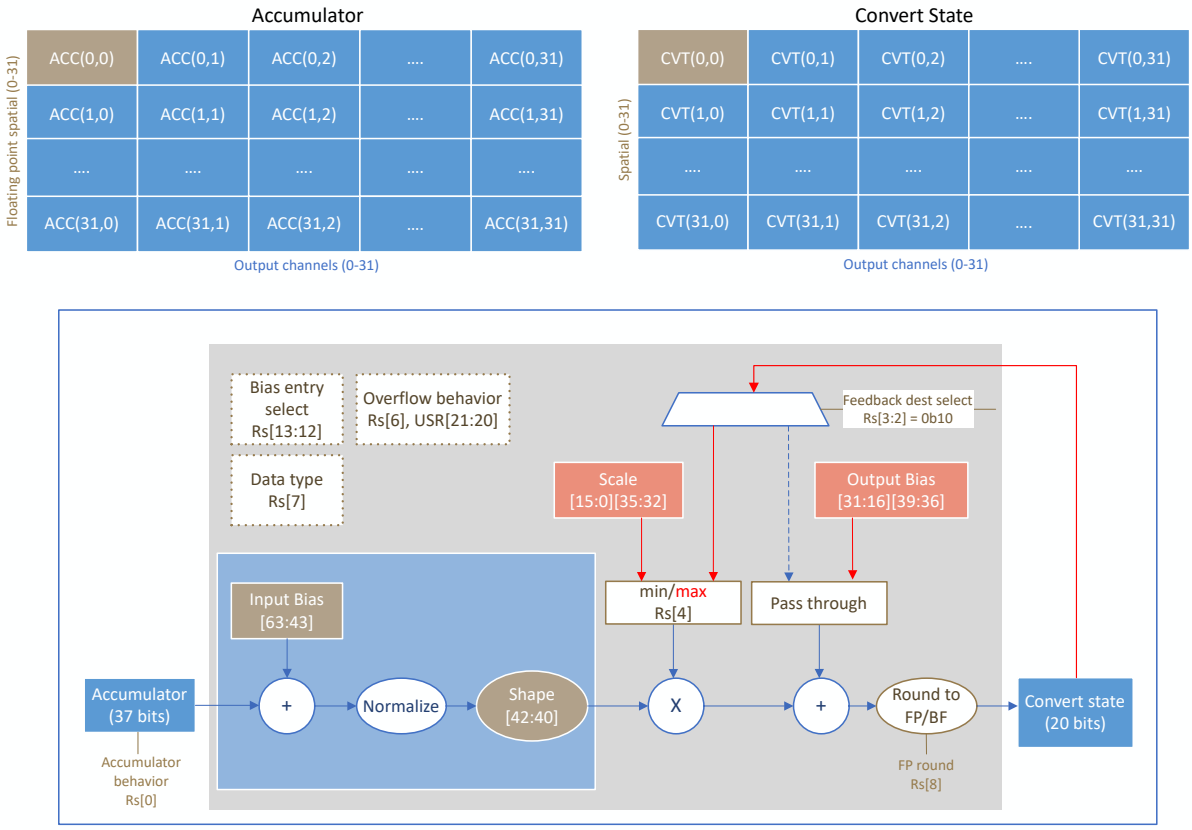

Figure 25. Convert transfer with feedback data

The figure above shows the instruction scalar register settings in red. In this case, the register selects the feedback to apply for scale (`Rs[fb_dest]=2`) and the max function (`Rs[fb_op]=1`) for comparison (Equation 3 in Table 9). The output bias is not selected for feedback, so the convert operation directly uses the output bias value from the bias register.

If the convert state selects the output bias as feedback (`Rs[fb_dest]=1`), then the min/max function is applied to the output bias selection and the convert operation directly uses the scale value from the bias register (Equation 1 or 2 in Table 9).

The first convert transfer instruction can set `Rs[Rnd]=1` such that there's four extra bits of precision stored in the convert state result for the next convert transfer's instruction feedback value. Note that HMX only stores FP16 to VTCM, the extra four bits are internal to the convert state and only used for feedback.

5.1. Instruction setup

For the first convert instruction prior to the feedback, set the `Rs[acc]` field in the convert transfer instruction as seen in Table 6 to keep the accumulator values from being cleared. Optionally, set the `Rs[Rnd]` for extra precision during feedback.

For the second convert instruction using the values of the prior instruction as feedback, set the `Rs[fb_dest]` and `Rs[fb_op]` fields. Ensure the correct bias register index is selected.

5.2. Code example

The example below follows the settings in the Figure 25 above.

It uses similar VTCM setup as Section 3.1 but with an additional bias register write to VTCM.

```
    int32_t *bias_start_addr = (int32_t*)((uintptr_t)wgt_start_addr +
wgt_size);
    int bias_size = 256;
    write_bias_vtcm(bias_start_addr);
    write_bias_vtcm((int32_t*)((uint8_t*)bias_start_addr + bias_size));
```

Replace the bias and convert transfer instruction register setup section in the code in Section 3.2 with the below code:

```
    // load bias first
    int32_t *bias_rs = bias_start_addr; // loading to bias index 0
    Q6_bias_mxmem2_A(bias_rs);

    int extra_precision = 1;  // save extra 4 bits in cvt state
    int retain_acc = 1;  // keep acc val for the next cvt insn
    uint32_t cvt_rs = (extra_precision << 8) | retain_acc;
    Q6_cvt_hf_acc_R(cvt_rs);

    // load bias for the 2nd convert insn
    int bias_idx = 1;
    bias_rs = (int32_t*)((uint8_t*)bias_start_addr + 256);
    bias_rs = (int32_t*)((uintptr_t)(bias_rs) | bias_idx);  // loading
to bias index 1
    Q6_bias_mxmem2_A(bias_rs);

    // 2nd convert insn use feedback from 1st convert insn
    cvt_rs = 0;
    int fb_dest = 2; // scale
    int fb_limit = 1; // max
    cvt_rs = (bias_idx << 12) | (fb_limit << 4) | (fb_dest << 2);
    Q6_cvt_hf_acc_R(cvt_rs);
```

Chapter 6. References

6.1. A.1 Acronyms and terms

Acronym or term	Definition
CISC	Complex instruction set computer
FP	Floating point
HMX	Hexagon Matrix Extensions
HVX	Hexagon Vector Extensions
MAC	Multiply accumulate
SIMD	Single instruction multiple data
VTCM	Vector tightly coupled memory

LEGAL INFORMATION

Your access to and use of this material, along with any documents, software, specifications, reference board files, drawings, diagnostics and other information contained herein (collectively this "Material"), is subject to your (including the corporation or other legal entity you represent, collectively "You" or "Your") acceptance of the terms and conditions ("Terms of Use") set forth below. If You do not agree to these Terms of Use, you may not use this Material and shall immediately destroy any copy thereof.

1) Legal Notice.

This Material is being made available to You solely for Your internal use with those products and service offerings of Qualcomm Technologies, Inc. ("**Qualcomm Technologies**"), its affiliates and/or licensors described in this Material, and shall not be used for any other purposes. If this Material is marked as "**Qualcomm Internal Use Only**", no license is granted to You herein, and You must immediately (a) destroy or return this Material to Qualcomm Technologies, and (b) report Your receipt of this Material to qualcomm.support@qti.qualcomm.com. This Material may not be altered, edited, or modified in any way without Qualcomm Technologies' prior written approval, nor may it be used for any machine learning or artificial intelligence development purpose which results, whether directly or indirectly, in the creation or development of an automated device, program, tool, algorithm, process, methodology, product and/or other output. Unauthorized use or disclosure of this Material or the information contained herein is strictly prohibited, and You agree to indemnify Qualcomm Technologies, its affiliates and licensors for any damages or losses suffered by Qualcomm Technologies, its affiliates and/or licensors for any such unauthorized uses or disclosures of this Material, in whole or part.

Qualcomm Technologies, its affiliates and/or licensors retain all rights and ownership in and to this Material. No license to any trademark, patent, copyright, mask work protection right or any other intellectual property right is either granted or implied by this Material or any information disclosed herein, including, but not limited to, any license to make, use, import or sell any product, service or technology offering embodying any of the information in this Material.

THIS MATERIAL IS BEING PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY LAW, QUALCOMM TECHNOLOGIES, ITS AFFILIATES AND/OR LICENSORS SPECIFICALLY DISCLAIM ALL WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, COMPLETENESS OR ACCURACY, AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MOREOVER, NEITHER QUALCOMM TECHNOLOGIES, NOR ANY OF ITS AFFILIATES AND/OR LICENSORS, SHALL BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY EXPENSES, LOSSES, USE, OR ACTIONS HOWSOEVER INCURRED OR UNDERTAKEN BY YOU IN RELIANCE ON THIS MATERIAL.

Certain product kits, tools and other items referenced in this Material may require You to accept additional terms and conditions before accessing or using those items.

Technical data specified in this Material may be subject to U.S. and other applicable export control laws. Transmission contrary to U.S. and any other applicable law is strictly prohibited.

Nothing in this Material is an offer to sell any of the components or devices referenced herein.

This Material is subject to change without further notification.

In the event of a conflict between these Terms of Use and the Website Terms of Use on www.qualcomm.com, the *Qualcomm Privacy Policy* referenced on www.qualcomm.com, or other legal statements or notices found on prior pages of the Material, these Terms of Use will control. In the event of a conflict between these Terms of Use and any other agreement (written or click-through, including, without limitation any non-disclosure agreement) executed by You and Qualcomm Technologies or a Qualcomm Technologies affiliate and/or licensor with respect to Your access to and use of this Material, the other agreement will control.

These Terms of Use shall be governed by and construed and enforced in accordance with the laws of the State of California, excluding the U.N. Convention on International Sale of Goods, without regard to conflict of laws principles. Any dispute, claim or controversy arising out of or relating to these Terms of Use, or the breach or validity hereof, shall be adjudicated only by a court of competent jurisdiction in the county of San Diego, State of California, and You hereby consent to the personal jurisdiction of such courts for that purpose.

2) Trademark and Product Attribution Statements.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. The Bluetooth® word mark is a registered trademark owned by Bluetooth SIG, Inc. Other product and brand names referenced in this Material may be trademarks or registered trademarks of their respective owners.

Snapdragon and Qualcomm branded products referenced in this Material are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.

THE DOCUMENTATION ACCOMPANYING THE MATERIALS AND/OR RELEVANT PRODUCTS AND SERVICE OFFERINGS MAY INCLUDE IMPORTANT USE LIMITATIONS. ANY DEVIATIONS FROM APPLICABLE USE LIMITATIONS MAY ADVERSELY IMPACT PERFORMANCE, DURABILITY, QUALITY OR SAFETY. YOU ASSUME ALL RISKS AND LIABILITIES ASSOCIATED WITH ANY DEVIATIONS.