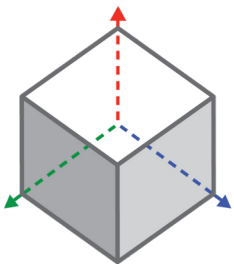

VectorBlox
embedded supercomputing

# VectorBlox MXP Programming Reference

August 31, 2015

VectorBlox
http://vectorblox.com

# Contents

# 1 MXP Processor

## 1.1 MXP Processor Architecture

The VectorBlox MXP Matrix Processor is an accelerator for data-parallel operations. It excels at applications like image processing where every pixel in a large set of data is subjected to the same sequence of calculations. With just a single instruction, the MXP can apply an operation to a vector, matrix, submatrix, volume, or sub-volume of data.

Architecturally, a vector engine and a DMA engine comprise the MXP processor. For maximum performance, the vector engine and the DMA engine can run at a different clock frequency than the host processor, or they can be frequency-locked with the host to provide guaranteed low-latency communication.

The MXP vector engine is made up of several parallel vector lanes. Each lane contains a 32-bit ALU and a portion of scratchpad memory. A typical MXP instance will have from 1 to 256 vector lanes, and roughly 4KB of scratchpad per vector lane. The ALU and scratchpad can also be subdivided to operate on sub-word arithmetic with halfwords (16-bits) or bytes (8-bits). In this case, the architecture provides two or four times the amount of data-level parallelism.

The MXP DMA engine can operate concurrently with the vector engine, allowing it to fetch operands or store results in main memory while vector calculations are taking place. For many compute-intensive algorithms, the latency of main memory can be completely hidden by the programmer through double buffering.

### 1.1.1 Scratchpad

Unlike traditional processors, the MXP does not have any data or address registers. Instead, all vector or matrix operations are performed *memory-to-memory* directly upon the scratchpad. This maximizes MXP performance, as the MXP does not need to load or store any vector data, and it does not need additional memory to hold a working set of values. From another viewpoint, the scratchpad *is* the register file.

The scratchpad works much like an explicitly managed cache. The primary means to transfer data in or out of the scratchpad is under DMA control. However, elements in the scratchpad can be directly accessed by the host processor by reading or writing a pointer into the scratchpad. This is considered a secondary access mechanism and is intended primarily for debugging or communicating very small amounts of information. For performance reasons, access to the scratchpad in this fashion is asynchronous with the vector engine, i.e. reads and writes to the pointer are not guaranteed to be executed in program order relative to vector instructions. To guarantee ordering, a programmer must explicitly stall the host processor to synchronize it with the MXP processor.

### 1.1.2 Other Processor State

In addition to the scratchpad, the MXP has 32 internal control registers, each of 32 bits in size. These are organized as a 32-entry memory, but they are not memory-mapped. To access them, an API is provided to save and restore each entry.

The first set of sixteen control registers are reserved for hardware use. They presently contain the mask status register, current vector length and parametersgoverning the 2D and 3D matrix sizes, for a total of 10 words. This leaves 6 words still undefined.

The second group of sixteen control registers are software-defined. Presently, only the first word of this set is defined – it contains an instance pointer to a `vbx_mxp_t` data structure which fully describes all parameters regarding the MXP processor hardware (such as the number of lanes and its clock frequency).

There is no other state information held in the MXP hardware.

### 1.1.3 VBX API and VBXware Software State

The VBX API maintains some global state to manage the scratchpad. In particular, it maintains a stack called *spstack* which can be used to save/restore the current position of scratchpad pointer. The scratchpad pointer is the high-water mark of memory allocated out of the scratchpad.

A future enhancement of the VBX API will use *spstack* to save and restore the processor state as well as the scratchpad pointer.

Another future enhancement of the VBX API will allow callees to spill scratchpad data to external memory when there is insufficient scratchpad space remaining. This will use the scratchpad in a LIFO fashion, spilling the beginning of the scratchpad once it is full. In this way, each layer (of a stack of library software) can potentially utilize the entire scratchpad for its own purposes before returning to the caller.

Each portion of the VBXware API maintains its own state information.

## 1.2 MXP Processor Parameters

Each MXP vector engine can be uniquely configured by several parameters, including the number of vector lanes or the size of the scratchpad. These parameters are maintained in the `vbx_mxp_t` data structure, which is defined in `vbx_types.h`. A pointer to this structure is stored in one of the control registers of the MXP processor. The sample code below shows how to obtain this instance pointer and access the parameters.

```
vbx_mxp_t * the_mxp = VBX_GET_THIS_MXP();
int num_vector_lanes = the_mxp->vector_lanes;
```

A list of the parameters stored in this structure and their meaning are given in the following table

| Type | Parameter | Description |
|---|---|---|
| *Fixed MXP CPU characteristics* | | |
| `vbx_void_t` | scratchpad_addr | Uncached pointer to the first byte of the scratchpad |
| `vbx_void_t *` | scratchpad_end | Uncached pointer one byte past the end of the scratchpad |
| `int` | scratchpad_size | Size of the scratchpad in bytes |
| `int` | core_freq | Clock frequency of the MXP in Hz |
| `short` | vector_lanes | Number of vector lanes |
| `short` | vcustomX_lanes | Number of lanes for `VCUSTOMX` (X from 0 to 15) opcode |
| `short` | max_masked_vector_length | Maximum length of a masked vector instruction |
| `short` | mask_partitions | Number of mask partitions (0 if disabled) |
| `char` | vector_custom_instructions | The number of vector custom instructions attached |
| `char` | fxp_word_frac_bits | Number of fractional bits in word-wise fixed-point multiply |
| `char` | fxp_half_frac_bits | Number of fractional bits in halfword-wise fixed-point multiply |
| `char` | fxp_byte_frac_bits | Number of fractional bits in byte-wise fixed-point multiply |
| *MXP flags* | | |
| `char` | init | A flag confirming this structure was initialized |
| *MXP run-time state* | | |
| `vbx_void_t *` | sp | Next free location in the scratchpad |

| Type | Parameter | Description |
|---|---|---|
| `vbx_void_t **` | spstack | Starting address of the stack of scratchpad pointers |
| `int` | spstack_top | Entry for the top-of-stack |
| `int` | spstack_max | Maximum number of words in the stack |

# 2 MXP Instruction Set

The MXP instruction set consists of a set of basic vector instructions. Furthermore, each basic vector instruction can be used in up to 432 different modes which govern further details about the operation, such as signed/unsigned, 1D/2D/3D, data size of bytes/halfwords/words, and whether to accumulate the results. These basic instructions and various modes are all described below.

## 2.1 Basic Vector Instructions

MXP vector instructions are invoked with the following macro:

```
vbx( mode, instr, dest, srcA, srcB );
```

The parameters to the macro are as follows:

- `mode` is a compile-time constant that provides type and datasize information
- `instr` is a compile-time constant representing the instruction
- `dest` is the destination operand
- `srcA` is the source operand A
- `srcB` is the source operand B

### 2.1.1 Source and Destination Operands

The operands have the following properties:

- `dest` is always a pointer to the scratchpad.
- `srcA` can be either a 32-bit scalar quantity (for scalar modes), or a pointer to the scratchpad (for vector modes).
- `srcB` is always a pointer to the scratchpad (for vector modes), or it is ignored (for the `VMOV` instruction, or enumerated vector modes). When ignored, we recommend specifying a `srcB` value of 0.

See the section on hazards regarding the possibility of data race hazards with vector instructions, particularly when operands point to overlapping memory regions.

### 2.1.2 Mode Type and Datasize Specifier

The `mode` is a 3 to 5 letter constant symbol where each letter has a meaning. The symbol can be generalized as a triplet `WXY` where:

- `W` is one of `{ VV, SV, VE, SE }`
- `X` is one of `{ B, H, W, BB, BH, BW, HB, HH, HW, WB, WH, WW }`
- `Y` is one of `{ S, U }` or empty (defaults to `S`).

The W field is a two-letter **type specifier** indicating vector (V), scalar (S), or enumerated (E) types. The first letter indicates the type for srcA, while the second letter indicates the type for srcB. Hence, srcA can only be a **vector** or **scalar type**, while srcB can only be a vector or **enumerated type**. The enumerated type produces a vector of constants for operand srcB, starting at 0 and incrementing by 1 for each vector element. Since enumerated types ignore the value specified in srcB, we recommend specifying a value of 0.

For 2D and 3D matrix instructions, discussed later, the enumerated value resets to 0 at the beginning of each new row. To produce totally unique enumerated elements across the entire matrix, additional instructions are required to add the row number or matrix number to each row or matrix.

The X field is a **datasize specifier** of byte (B), halfword (H), or word (W). When a single letter is specified, the source and the destination operands all use the specified datasize. The two-character versions with a repeated letter (namely BB, HH, and WW) are equivalent to B, H, and W, respectively.

When two different letters are used in the **datasize specifier**, a **datasize conversion** takes place. The first letter indicates the source datasize, while the second letter indicates the destination datasize. Before performing the requested operation, the processor will either expand the data (using sign-extension for signed, or zero-extension for unsigned) or truncate it to match the *larger* of the source and destination datasizes. For example, VVBH will take two source operands which are byte vectors, convert them to halfword input vectors on the fly, and compute a halfword vector result. This is convenient with addition, subtraction, and multiply instructions, as it eliminates the possibility of overflow/carry-out. It is also possible to go in the reverse direction, e.g. VVHB, which combines two halfword vectors and truncates the result into a byte vector. A datasize reduction such as this may be useful if the difference between vectors is guaranteed to be small. Note that both source operands, srcA and srcB, must have matching datasizes. Scalar and enumerated operands are not truncated to the input width when expanding (BH, BW, or HW datasize specifier), but rather used at the larger output width.

There are two special cases to consider with **datasize conversion**. First, datasize conversion with the fixed-point multiply instruction, VMULFXP, produces an undefined result. To convert between fixed-point representations (byte/halfword/word) where the binary point is in a different position, use a shift operation (VSHR or VSHL) to move the binary point and a datasize conversion to expand or truncate the result. Second, datasize conversion with an accumulated vector instruction performs the requested operation at the original source datasize, not at the larger datasize. However, the destination datasize is used when accumulating the intermediate values and writing the final result.

The Y field is an optional **sign specifier** of unsigned (U) or signed (S). If omitted, the default behaviour is always *signed*. Note that logical versus arithmetic right-shifting via the VSHR instruction is distinguished via this signed specifier. Also, note that some instructions , for example the bit-rotating instructions VROTR and VROTL, don't make sense as signed operations and so behave as an *unsigned* operation no matter which **sign specifier** is used.

### 2.1.3 Instruction Specifier

The instr is a constant symbol specifying the desired vector operation. The instructions must be one of:

- bitwise logical VAND, VOR, VXOR, VSHL, VSHR, VROTL, VROTR
- arithmetic VADD, VSUB, VADDC, VSUBB, VABSDIFF, VMUL, VMULLO, VMULHI, VMULFXP
- move and conditional-move VMOV, VCMV_LEZ, VCMV_GTZ, VCMV_LTZ, VCMV_GEZ, VCMV_Z, VCMV_NZ
- custom VCUSTOMX, for X from 0 to 15

For the shift and rotate instructions, srcA specifies the amount of the shift/rotate, while srcB specifies the value to be shifted/rotated. The sign specifier distinguishes arithmetic from logical shifts.

For the unconditional move instruction, VMOV, operand srcB is unused and should be specified as 0.

The arithmetic instructions ADDC and SUBB are used for extended-precision operations where the carry/borrow flag from the srcB operand is added to the srcA operand.

For multiplication, the result requires twice as many bits as the source operands. Hence, the operations `VMUL` or `VMULLO` produce the lower half of the product, while `VMULHI` produces the upper half of the product. Multiplication can also be used with a datasize-conversion operation from byte-to-halfword, or from halfword-to-word (see the section on datasize specifiers). In this case, the `VMUL` or `VMULLO` instruction places the entire product in the destination.

For instructions that undergo a datasize conversion, execution at the ALU is always performed at the widest datasize, unless an accumulated instruction is done in which case the source datasize is used. Expansion or up-conversion relies upon sign-extension (for signed) or zero-extension (for unsigned). Conversely, down-conversion relies upon truncation, with the upper bytes being discarded.

Many instructions can also generate a 1-bit flag result per element. Also, a few instructions alter their behaviour depending upon the flag setting. These are discussed in the section on conditional moves.

The fixed-point multiply operation `VMULFXP` will perform a multiply followed by a shift right to keep the correct number of fractional bits when multiplying fixed-point numbers. The fixed-point format, and therefore the amount of the right shift, is fixed and predetermined at system build time. Formats with no integer bits (Q0.8 for byte, Q0.16 for halfword, and Q0.32 for word) are not supported by `VMULFXP`; `VMULHI` performs the function a `VMULFXP` operation would for these formats and should be used instead. `VMULFXP` may overflow if the result is too large in magnitude to fit in the supported fixed-point format. In the case of overflow, the flag bit is set to 1 (see updates to flag state). Additionally, signed `VMULFXP` operations are sign preserving; the MSB of the result is set to the numerically correct sign even if the operation overflowed, so that the result can be saturated positive or negative if desired. The `VMULFXP` instruction produces an undefined result when used together with datasize conversion.

For the conditional move instructions, `srcA` provides the value to move, while `srcB` provides the logical predicate. For each element of a conditional move, a logical true predicate value results in the value from `srcA` being written to `dest`. Further details concerning conditional moves and flags are discussed in the section on conditional moves.

Custom instructions (`VCUSTOM0-15`) are an experimental feature allowing system specific instructions through an external port to the MXP. MXP instances may have 0 to 16 custom operators attached. If a custom instruction is issued while there is no custom operator on the corresponding port, the result is undefined. Simple custom instructions function the same way as normal vector instructions, while more complex instructions can be built that allow for more than two inputs and multiple outputs but require extra setup. The number of custom instruction lanes may be equal to or fewer than the number of vector lanes; this can be queried by software reading the `vcustomX_lanes` parameter corresponding to the `VCUSTOMX` opcode being used. In the case the number of custom instruction lanes is fewer than the number of vector lanes, custom instructions will take correspondingly longer to execute. Consult your system documentation to find the custom instructions it has attached and their usage. Information for custom instruction creators can be found in the *VectorBlox MXP Custom Instruction Manual*.

### 2.1.4 Specifying 1D Vector Length

Before invoking any 1D vector instruction, the programmer must first specify the dimensions of the data to the MXP. This is done by setting the *vector length register* using:

```
vbx_set_vl( vl );
```

The *unsigned* integer argument `vl` specifies the number of **vector elements** in the operation. Hence, a setting of `vl = 10` will operate on 10 bytes, 10 halfwords, or 10 words, depending upon the datasize specifier. The number of **vector elements** can be set from 1 to the full vector scratchpad size (in bytes), inclusive. Setting a vector length of 0, or greater than the scratchpad size, results in undefined behaviour.

The MXP will always remember the latest vector length that was set. While this is convenient, it can also be problematic if called functions or third party libraries change the value as a side effect. To reduce such unpleasant side effects, the VBXware library always saves and restores the vector length register (as well as the 2D and 3D dimension registers).

## 2.2 Accumulated Vector Instructions

All basic MXP instructions can be compounded with an **accumulate** operation. This accumulation does not require any additional run-time. To use it, replace the `vbx()` macro with the `vbx_acc()` macro. The arguments for this macros are exactly the same as before:

```
vbx_acc( mode, instr, dest, srcA, srcB );
```

The **accumulate** operation will sum all of the elements of a vector. Hence, the result written to `dest` is a single element, not a vector of elements. The size of the result element is determined by the destination datasize specifier.

When used with the `VABSDIFF` instruction, the **accumulate** operation computes the sum-of-absolute-differences in a single instruction as follows:

```
vbx_acc( mode, VABSDIFF, dest, srcA, srcB );
```

## 2.3 2D and 3D Matrix Instructions

Vector instructions can operate on 1D vectors as well as 2D or 3D matrices. The 2D and 3D modes provide higher performance and increased code density.

The following API calls are used to invoke 2D or 3D operations:

```
vbx_2D()

vbx_acc_2D()

vbx_3D()

vbx_acc_3D()
```

To use them, simply replace the 1D vector operation `vbx()` with `vbx_2D()` or `vbx_3D()`, for example. The arguments for these macros are exactly the same as before.

### 2.3.1 Specifying 2D and 3D Matrix Sizes

Before invoking a 2D matrix instruction, the programmer must set the current *2D matrix size registers* using two calls:

```
vbx_set_vl( vl );
vbx_set_2D( numRows, incDest2, incSrcA2, incSrcB2 );
```

Likewise, before invoking a 3D matrix instruction, the programmer must set the current *3D matrix size registers* using three calls:

```
vbx_set_vl( vl );
vbx_set_2D( numRows, incDest2, incSrcA2, incSrcB2 );
```

```
vbx_set_3D( numMats, incDest3, incSrcA3, incSrcB3 );
```

These functions are further described in the section on vector length and matrix size.

For both 2D and 3D operations, the increment amounts for the sources and destination are specified in units of bytes, not elements. This differs from the `vbx_set_vl()` function which specifies the vector length in units of elements.

The MXP will always remember the latest vector length and matrix dimensions that were set. While this is convenient, it can also be problematic if called functions or third party libraries change the values as a side effect. To reduce such unpleasant side effects, the VBXware library always saves and restores these registers.

### 2.3.2 Accumulating 2D and 3D Matrices

When using the *accumulate* compound operation with 2D matrix operations, the MXP will write one result element per row. This produces a vector of result elements equal in length to `numRows`. The size of each result element is determined by the destination datasize specifier.

Likewise, 3D matrix operations will write one vector of result elements per submatrix, producing a matrix of result elements of width `numRows` and of height `numMats`.

## 2.4 Conditional Move and Flags

Data-dependent behaviour within the vector engine is accomplished via conditional move instructions. This allows you to implement thresholding and other types of conditional behaviour.

In addition to the normal vector data of 8, 16, or 32 bits per element, MXP stores one additional bit per element, called a **flag**. This bit can be set or cleared by certain operations, and it can alter the behaviour of other operations.

In particular, the flag is used by relational conditional move instructions, along with the regular of the bits in each element, to calculate the move predicate. Also, the `VADDC` and `VSUBB` instructions consider the flag as extra bit for carry-in or borrow.

When the flag bit of a scalar inputs is used, its flag is treated being $0$. That is, for `SV` and `SE` type specifiers, $F_A$ is $0$. Likewise enumerated inputs are treated as $0$; for `VE` and `SE` type specifiers $F_B$ is $0$.

### 2.4.1 Updates to Flag State

An element's flag bit is modified by the operations listed in

| Sign specifier | Instruction | Flag produced |
|---|---|---|
| Unsigned | any *accumulate* | $V$ (subject to change) |
| Signed | any *accumulate* | $V\dagger$ (subject to change) |
| Unsigned | `VADD` | $C$ |
| Unsigned | `VADDC` | $C$ |
| Unsigned | `VSUB` | $B$ |
| Unsigned | `VSUBB` | $B$ |
| Signed | `VADD` | $V$ |
| Signed | `VADDC` | $V$ |
| Signed | `VSUB` | $V$ |

| Sign specifier | Instruction | Flag produced |
|---|---|---|
| Signed | `VSUBB` | $V$ |
| Either | `VMUL` | $V$ |
| Either | `VMULLO` | $V$ |
| Either | `VSHL` | $V$ |
| Either | `VMULHI` | $R$ |
| Either | `VSHR` | $R$ |
| Unsigned | `VMULFXP` | $V$ (subject to change) |
| Signed | `VMULFXP` | $V\dagger$ (subject to change) |
| Either | `VROTL` | $F_B$ (i.e., copy from source B) |
| Either | `VROTR` | $F_B$ (i.e., copy from source B) |
| Either | `VMOV` | $F_A$ (i.e., copy from source A) |
| Either | `VCMV_*` | $F_A$ (if predicate B is true, else unchanged) |
| Either | `VAND` | $F_A$ & $F_B$ |
| Either | `VXOR` | $F_A \oplus F_B$ |
| Either | `VOR` | $F_A \mid F_B$ |
| Either | `VABSDIFF` | $0$ (subject to change) |
| Either | any `vbx_dma_to_vector()` | $0$ |

> Key: $C$ carry-out $B$ borrow $R$ rounding bit is located one rank position lower than the LSB of the final result $V$ overflow; for left-shift and multiply-low, set if any bits of significance are lost † sign preserving; sets the MSB to the sign of the internal full precision result $F_A$ flag of source operand A $F_B$ flag of source operand B

Any operation which includes an **accumulate** will have its flag set to the accumulation overflow value. Accumulation, regardless of operand sizes, happens internally with 40 bits of precision. The overflow flag is set to 1 if the final 40-bit internal accumulator value has any bits of significance that do not fit in the 32-bit instruction result. Additionally, 32-bit signed accumulates are sign preserving: the MSB of the result is set to the MSB of the 40-bit internal accumulator. 16-bit and 8-bit signed accumulates are truncated from the 32-bit value (the sign preservation is lost), so when saturating a signed accumulate operation an output precision of 32-bits must be used. Because the 40-bit internal accumulator itself can overflow, when checking for accumulation overflow on 32-bit inputs it is recommended to break the accumulation in to chunks of size 256 or less (as 256 32-bit inputs cannot overflow a 40-bit accumulator) and check for overflow on each chunk.

Overflow under a left shift operation requires some explanation. An unsigned `VSHL` produces an overflow if any of the bits shifted out are set. Likewise, a signed `VSHL` produces an overflow if an originally positive value shifts out any set bits, or if an originally negative value shifts out any cleared bits.

Operations which include a datasize conversion operate as follows. In an up-conversion (expansion), any vector source operands are first extended to the wider size before applying the operator. Scalar and enumerated operands are not truncated to the input width, but rather the wider of the two widths. In a down-conversion (truncation), the operation is done on the wider datasize and the final result is truncated. In both cases, the flag is derived from the wide result according to the operation performed.

### 2.4.2 Operations using Flag State

The flag is used by the following instructions:

- The `VADDC` instruction treats the flag as a carry-in bit, adding it to the addend
- The `VSUBB` instruction treats the flag as a borrow bit, subtracting it from the minuend (left operand)
- The `VCMV_LTZ`, `VCMV_LEZ`, `VCMV_GTZ`, and `VCMV_GEZ` instructions use the flag to properly correct for overflow (signed) and carry/borrow (unsigned) conditions

Conditional move predicates are computed according to

| Sign specifier | Instruction | Predicate expression |
|---|---|---|
| Unsigned | `VCMV_FS` | $F$ |
| Unsigned | `VCMV_FC` | $\neg F$ |
| Signed | `VCMV_FS` | *undefined* |
| Signed | `VCMV_FC` | *undefined* |
| Unsigned | `VCMV_LTZ` | $F$ |
| Unsigned | `VCMV_GEZ` | $\neg\ F$ |
| Unsigned | `VCMV_LEZ` | $F \mid Z$ |
| Unsigned | `VCMV_GTZ` | $\neg$ ( $F \mid Z$ ) |
| Signed | `VCMV_LTZ` | $F \oplus N$ |
| Signed | `VCMV_GEZ` | $\neg\ (F \oplus N)$ |
| Signed | `VCMV_LEZ` | $(F \hat{}\ N) \oplus Z$ |
| Signed | `VCMV_GTZ` | $\neg$ ( $(F \oplus N) \mid Z$) |
| Either | `VCMV_Z` | $Z$ |
| Either | `VCMV_NZ` | $\neg Z$ |

> Key: $Z$ indicates all bits of the element (excluding the flag) are zero $N$ is the MSB of the element $F$ is the flag value of the element The predicate expression depends only upon source operand B.

### 2.4.3 Flag Examples

Understanding operation of the flags and conditional move can be helped by examples. For example, the following code fragment saturates values in a vector to +100:

```
vbx( SVB, VSUB,     v_sub, 100, v_val );
vbx( SVB, VCMV_LTZ, v_val, 100, v_sub );
```

These instructions can be read as:

```
for( i=0; i<vl; i++ ){
  v_sub[i] = 100 - v_val[i];
}
for( i=0; i<vl; i++ ){
  if( v_sub[i] < 0 ){
    v_val[i] = 100;
  }
}
```

Keep in mind that the predicate expression in the `if( ... )` also uses the flags stored in the `v_sub` array. Note the precise `mode` (type, datasize and sign specifiers) must match between the `VSUB` which computes the flags and the `VCMV` which uses the flags.

Another example takes two vectors and produces one vector with the smaller (minimum values) and another vector with the larger (maximum) values:

```
vbx( VVBU, VMOV,     v_tmp,  v_min,     0 );
vbx( VVBU, VSUB,     v_sub,  v_max, v_min );
vbx( VVBU, VCMV_LTZ, v_min,  v_max, v_sub );
vbx( VVBU, VCMV_LTZ, v_max,  v_tmp, v_sub );
```

These instructions can be read as:

```
for( i=0; i<vl; i++ ) v_tmp[i] = v_min[i];
for( i=0; i<vl; i++ ) v_sub[i] = v_max[i] - v_min[i];
for( i=0; i<vl; i++ ) if( v_sub[i] < 0 ) v_min[i] = v_max[i];
for( i=0; i<vl; i++ ) if( v_sub[i] < 0 ) v_max[i] = v_tmp[i];
```

In this example, the same set of flags is used more than once.

Finally, flags can be combined with accumulated vector instructions see accum. The example below counts the number of entries which are smaller than or equal to the saturation threshold of +100:

```
vbx(     SVB, VSUB,     v_sub,  100, v_val );
vbx_acc( SVB, VCMV_GEZ, v_val,    1, v_sub );
```

These instructions can be read as:

```
for( i=0; i<vl; i++ ) v_sub[i] = 100 - v_val[i];
accum=0;
for( i=0; i<vl; i++ ) if( v_sub[i] >= 0 ) { accum+=1; }
v_val[0] = accum;
```

This is a rather advanced example. It works because the scalar value used for all elements in source operand A are only valid (i.e., enabled for accumulation) if the predicate expression is true; a predicate expression of false disables accumulation of that element, effectively adding a value of zero.

## 2.5 Masked Vector Instructions

MXP can support an additional data-dependent execution through the use of masked vector instructions. When executing, a masked vector instruction will skip wavefronts that would not cause any data to be written back. A

wavefront is one cycle worth of data; on a V4 with a word vector elements 0 to 3 would be one wavefront, elements 4 to 7 another. By skipping unused wavefronts execution time can be shortened, especially if the data to be process is mostly masked off. In contrast to a conditional move operation, it requires a setup instruction to create the mask that will be used on subsequent instructions. This means masked vector instructions should be used only when the mask is reused multiple times; however, if many instructions are processed and the mask is sparse, the savings from skipping unused wavefronts can be large.

Masked instructions are disabled when the `mask_partitions` parameter is zero. Currently MXP only supports 0 (disabled) or 1 mask partition. Multiple mask partitions are a future enhacement that will allow partial wavefronts to skip independently; for instance, a V4 with two partitions would let lanes 0 and 1 skip to a different place within the vector than lanes 2 and 3. The other parameter controlling masked instructions is the `max_masked_vector_length`, or MMVL. The MMVL is a limitation on how long a mask vector can be, and is nomally set to be smaller than the full scratchpad depth to save resources. When using masked instructions, the programmer must check that the vector length used to set up the mask is less than the masked vector length.

Note that masked vector instructions do not support enumerated (`SE`/`VE`) modes. If an enumerated operand is needed in a masked instruction, an enumerated vector must be created in scratchpad using a non-masked instruction and then used in a vector (`SV`/`VV`) mode masked instruction.

To use a masked vector instruction, first the mask must be set:

```
vbx_setup_mask( mode, mask_test, src );
```

`mode` is the same as for other operations, selecting the datasize and if the test is signed or unsigned. `mask_test` is one of the `VCMV_*` tests (see Conditional Move and Flags). `src` is the vector to be tested; if the `VCMV` is true and would write back, that bit is set in the MXP's internal mask state, else it is not set and can be skipped if in a wavefront without any other valid mask bits. This mask is then valid for all subsequent masked instructions until the next `vbx_mask_setup()` instruction is issued.

Masked instructions take the form:

```
vbx_masked( mode, instr, dest, srcA, srcB );
```

They execute the same as normal instructions, except they do not write to elements that were masked off in the last `vbx_setup_mask()` instruction, and will skip any wavefront with no valid elements. If a mask is currently being used and it is desirable to do a `vbx_setup_mask()` test on only the subset of elements that are currently valid, the programmer can do a masked setup instruction:

```
vbx_setup_mask_masked( mode, mask_test, src );
```

### 2.5.1 Masked Instruction Example

The following shows an example where all non-zero elements in a vector are incremented then multiplied by three. This could be done with conditional move instructions by performing the increment and multiply in a temporary vector then writing the result back using the conditional move instruction. The masked version, by contrast, does not require a temporary vector and will execute in fewer cycles if the data is mostly zero.

```
vbx_setup_mask( SVW, VCMV_NZ, v_a);
vbx_masked( SVW, VADD, v_a, 1, v_a );
vbx_masked( SVW, VMUL, v_a, 3, v_a );
```

### 2.5.2 Mask Status Register

A mask status register is available to allow the programmer to check if the entire mask is empty, in which case all wavefronts are skipped. In this case execution will still proceed correctly, but since nothing is written back it may be faster to check if the mask is empty and then not dispatch the masked vector instructions at all. The mask status register is accesed via the function:

```
vbx_get_mask_status(int *mask_status);
```

Getting the mask status is non-blocking; if bit 31 is '1' then the result is invalid. Reading the mask status sets bit 31 to be invalid again. Bit 31 is cleared when a `vbx_setup_mask()` or `vbx_setup_mask_masked()` completes within the vector pipeline. Bits 30 to 0 return the mask status; if they are all '0' then the mask is empty. Currently any nonzero value only means the mask is not empty, but different return values are reserved for future use.

Though the call is non-blocking, it is simpler and easier to use in a blocking manner. One way is to preface the call with a `vbx_sync()` instruction to make sure all mask setup instructions have finished. The following example shifts every element of `v_a` left by 1 bit until the MSB of the element is a '0'.

```
int mask_status;

//First iteration, before mask has been setup
vbx( SVW, VAND, v_flag, 0x80000000, v_a );
vbx_setup_mask( SVW, VCMV_NZ, v_a );
vbx_masked( SVW, VSHL, v_a, 1, v_a );
vbx_sync();
vbx_get_mask_status( &mask_status );

//if mask_status is 0, the mask is empty and we are done
while( mask_status ){
  vbx_masked( SVW, VAND, v_flag, 0x80000000, v_a );
  vbx_setup_mask_masked( SVW, VCMV_NZ, v_a );
  vbx_masked( SVW, VSHL, v_a,             1, v_a );

  vbx_sync();
  vbx_get_mask_status( &mask_status );
}
```

Alternately, the code can poll on the invalid bit of the mask status register (bit 31) to check if the result is valid:

```
int mask_status;

//Set the mask status to invalid;
//Can be skipped if it's already known to be invalid
vbx_sync();
vbx_get_mask_status( &mask_status );

//First iteration, before mask has been setup
vbx( SVW, VAND, v_flag, 0x80000000, v_a );
vbx_setup_mask( SVW, VCMV_NZ, v_a );
vbx_masked( SVW, VSHL, v_a, 1, v_a );
//Spin until mask_status bit 31 is '0' (mask_status is valid)
do {
  vbx_get_mask_status( &mask_status );
} while ( mask_status & 0x80000000 );
```

```
while ( mask_status ) {
  vbx_masked( SVW, VAND, v_flag, 0x80000000, v_a );
  vbx_setup_mask_masked( SVW, VCMV_NZ, v_a );
  vbx_masked( SVW, VSHL, v_a,             1, v_a );

  //Spin until mask_status bit 31 is '0' (mask_status is valid)
  do {
    vbx_get_mask_status( &mask_status );
  } while ( mask_status & 0x80000000 );
}
```

The main difference between this version and the previous version is that the `vbx_sync()` call forces the scalar core to stall until ALL previous vector instructions have finished, whereas polling on the mask status register just requires waiting until the last mask setup instruction has finished. This means that when the do-while loop finishes on the scalar core, the previous `vbx_masked()` instruction may be executing on the vector core.

A further optimization is to use the mask status register in a non-blocking manner. Whether or not we use the mask status register to skip an iteration does not affect the functionality of the program; if we don't skip an iteration when the mask is empty all masked operations will effectively be NOPs. So instead of waiting for the current mask setup instruction to finish, we can just check the status of the last mask setup instruction to have finished within the vector core. Once the scalar core sees that the mask is empty it can exit the loop. It may have dispatched additional masked instructions, but they will have no effect since the mask will be empty when they execute and so nothing will be written back.

```
int mask_status;

//Set the mask status to invalid;
//Can be skipped if it's already known to be invalid
vbx_sync();
vbx_get_mask_status( &mask_status );

//First iteration, before mask has been setup
vbx( SVW, VAND, v_flag, 0x80000000, v_a );
vbx_setup_mask( SVW, VCMV_NZ, v_a );
vbx_masked( SVW, VSHL, v_a, 1, v_a );
vbx_get_mask_status( &mask_status );

while ( mask_status ) {
  vbx_masked( SVW, VAND, v_flag, 0x80000000, v_a );
  vbx_setup_mask_masked( SVW, VCMV_NZ, v_a );
  vbx_masked( SVW, VSHL, v_a,             1, v_a );

  //Get the latest mask status available
  vbx_get_mask_status( &mask_status );
}
```

## 2.6 C++ Equivalent Instructions

Vectorblox additionally provides the following c++ functions that simplify programming by inferring type data from the operands used:

```
    vbxx(instr,dest,srcA,srcB);

    vbxx_acc(instr,dest,srcA,srcB);
```

```
    vbxx_2D(instr,dest,srcA,srcB);

    vbxx_acc_2D(instr,dest,srcA,srcB);

    vbxx_3D(instr,dest,srcA,srcB);

    vbxx_acc_3D(instr,dest,srcA,srcB);
```

### 2.6.1 Example

```
//This function adds two vectors in the c manner
void add_c(){
    vbx_word_t* va=vbx_sp_malloc(100);
    vbx_word_t* vb=vbx_sp_malloc(100);
    bx_word_t* vc=vbx_sp_malloc(100);
    //add va to vb, and store in vc
    vbx(VVWS,VADD,vc,va,vb);
}

//This function adds two vectors in the c++ manner
void add_cxx(){
    vbx_word_t* va=vbx_sp_malloc(100);
    vbx_word_t* vb=vbx_sp_malloc(100);
    vbx_word_t* vc=vbx_sp_malloc(100);
    //add va to vb, and store in vc
    vbxx(VADD,vc,va,vb);
}
```

These c++ functions provide the same functionality as their C equivalents, but without the need to provide a mode parameter. Another thing to note is that the move instruction no longer needs the srcB parameter, it is just `vbxx(VMOV,dest,srcA)`

# 3 VBX API Library

The VBX API consists of a number of basic calls which are described in this section. However, it is important to note that many of these API calls require initialization. This initialization is done by a call to the `_vbx_init()` function, which is normally called automatically as part of the pre-initialization process before `main()` is started. (Calling this function more than once is not encouraged, as it may lead to an inconsistent state.)

## 3.1 Specifying Vector and Matrix Sizes

The APIs to set or retrieve vector and matrix sizes are:

```
vbx_set_vl( vl );

vbx_set_2D( numRows, incDest2, incSrcA2, incSrcB2 );

vbx_set_3D( numMats, incDest3, incSrcA3, incSrcB3 );

vbx_get_vl( &vl );

vbx_get_2D( &numRows, &incDest2, &incSrcA2, &incSrcB2 );

vbx_get_3D( &numMats, &incDest3, &incSrcA3, &incSrcB3 );
```

Notes:

- For normal vector (i.e., 1D) operations, the *unsigned* `vl` parameter to the `vbx_set_vl()` function specifies the length of a vector in units of elements.
- For 2D and 3D operations, the *signed* parameters `incDest2`, `incDest3`, `incSrcA2`, `incSrcA3`, `incSrcB2`, and `incSrcB3` are specified in units of bytes, not elements. These represent the stride, or distance in bytes, between one row to the next row (2D), or between one matrix to the next matrix (3D).

The 2D matrix operations are performed on rows of vectors, where each vector represents one row. The *unsigned* `numRows` parameter specifies the number of rows for which the vector operation should be repeated. While a 2D instruction is being executed, the destination initially specified by `dest` will be incremented by the amount `incDest2` after each row is completed. Hence, the `incDest2` value indicates the number of bytes per row in the destination matrix. It is quite common for the vector length (which is specified in units of elements, not bytes), to match the increment amount (after correcting for data size). However, it is also perfectly valid to use a longer or shorter vector length than the increment amount. For example, this behaviour can be used to implement a sliding-window effect, or to operate on a sub-matrix.

Likewise, after each row, the `srcA` operand will be internally incremented by the amount `incSrcA2`, and the `srcB` will be internally incremented by the amount `incSrcB2`.

Note that after the 2D instruction has finished executing, its actual operands, namely `dest`, `srcA` and `srcB`, are left unmodified.

For example, the following 2D program fragment adds two matrices: An equivalent C program would be:

1D Vector VBX code:

```
vbx_half_t *vdest, *vsrc1, *vsrc2;
vbx_set_vl( vl );
vbx( VVH, VADD, vdest, vsrc1, vsrc2 );
```

Equivalent C code:

```
for( c=0; c < vl; c++ ) {
    vdest[c] = vsrc1[c] + vsrc2[c];
}
```

2D Matrix VBX code:

```
    vbx_half_t *vdest, *vsrc1, *vsrc2;
    vbx_set_vl( vl );
    vbx_set_2D( numRows, iD2, iA2, iB2 );
    vbx_2D( VVH, VADD, vdest, vsrc1, vsrc2 );
```

Equivalent C code:

```
vbx_half_t *dest, *srcA, *srcB;
for( r = 0; r < numRows; r++ ) {
    dest = (vbx_half_t*)( (vbx_byte_t*)vdest + (r*iD2) );
    srcA = (vbx_half_t*)( (vbx_byte_t*)vsrc1 + (r*iA2) );
    srcB = (vbx_half_t*)( (vbx_byte_t*)vsrc2 + (r*iB2) );
    for( c=0; c < vl; c++ ) {
        dest[c] = srcA[c] + srcB[c];
    }
}
```

3D Matrix VBX code:

```
vbx_half_t *vdest, *vsrc1, *vsrc2;
vbx_set_vl( vl );
vbx_set_2D( numRows, iD2, iA2, iB2 );
vbx_set_3D( numMats, iD3, iA3, iB3 );
vbx_3D( VVH, VADD, vdest, vsrc1, vsrc2 );
```

Equivalent C code:

```
vbx_half_t *dest, *srcA, *srcB;
for( m = 0; m < numMats; m++ ) {
    for( r = 0; r < numRows; r++ ) {
        dest = (vbx_half_t*)( (vbx_byte_t*)vdest + (m*iD3) + (r*iD2) );
        srcA = (vbx_half_t*)( (vbx_byte_t*)vsrc1 + (m*iA3) + (r*iA2) );
        srcB = (vbx_half_t*)( (vbx_byte_t*)vsrc2 + (m*iB3) + (r*iB2) );
        for( c=0; c < vl; c++ ) {
            dest[c] = srcA[c] + srcB[c];
        }
    }
}
```

Note that it is possible to specify increments of 0, which implies the same vector will be re-used for each row. This allows a vector to be added row-wise to a matrix, for example. Also, negative increments can be used to iterate 'backwards' through a matrix, starting with the last row. Finally, unit increments are also possible, providing a sliding-window effect on a vector.

The 3D matrix operations work exactly like the 2D operations, but the increment amounts are added after each 2D matrix is completed.

In all cases, the original operands are left unmodified; all incrementing takes place in internal registers of the MXP CPU.

The `get` functions allow the programmer to query the current set of parameters from the hardware. These are used in subroutines which need to transparently save (`get`) and restore (`set`) these parameters without disturbing the caller. In all cases, a pointer to an integer is passed for each argument of the `get` routines.

## 3.2 DMA Operations

The DMA operation APIs are:

```
vbx_dma_to_host( hostPtr, scratchPtr, numBytes );

vbx_dma_to_vector( scratchPtr, hostPtr, numBytes );

vbx_dma_to_host_2D( hostPtr, scratchPtr, rowLengthBytes, numRows,
                    hostInc, scratchInc );

vbx_dma_to_vector_2D( scratchPtr, hostPtr, rowLengthBytes, numRows,
                      scratchInc, hostInc );

vbx_sync();
```

These operations initiate a hardware-accelerated block asynchronous copy operation from the scratchpad to the host, or from the host memory to the scratchpad. DMA operations from scratchpad to scratchpad or from host memory to host memory are not supported.

DMA operations issue in program order, and are placed into a 2-entry DMA queue. If the queue is currently full while a DMA operation is trying to issue, the processor will stall until the current DMA operation finishes. Once placed into the queue, the vector processor core resumes executing vector instructions concurrently. In the case where a vector instruction has a data race hazard with an enqueued DMA operation, the vector processor will stall. By moving DMA operations as early as possible in the code, stalling can be avoided. Please see the section on hazards for more information on data race hazards between DMA operations and vector instructions, and on the use of the `vbx_sync()` function.

For repeated DMA operations on regular data, such as loading on storing a sub-matrix, the 2D DMA operations can be used. These function similar to 2D instructions, issuing repeated DMA operations with a source and destination increment between each. 2D DMA operations are faster than issuing repeated 1D DMA operations and only take one entry in the DMA queue, so they are preferred where possible.

2D DMA operation:

```
vbx_byte_t *vdest, *src;
vbx_dma_to_vector_2D( vdest, src, rowLengthBytes, numRows, scratchInc, hostInc );
```

Equivalent C code:

```
vbx_byte_t *vdest, *src;
for( r = 0; r < numRows; r++ ) {
    for( c = 0; c < vl; c++ ) {
        vdest[c + (r*scratchInc)] = src[c + (r*hostInc)];
    }
```

```
}
```

## 3.3 Scratchpad Management

The scratchpad management APIs are:

```
vbx_sp_malloc( numBytes );

vbx_sp_free();

vbx_sp_push();

vbx_sp_pop();

vbx_sp_get();

vbx_sp_set( newSP );
```

The scratchpad is used like a stack, similar to a subroutine allocating and de-allocating memory from the scratchpad as needed. These functions are used to manage allocation and deallocation from the scratchpad. The functions are intended to execute quickly, hence the stack-like model (to avoid garbage collection and fragmentation issues).

The `vbx_sp_malloc()` function returns a pointer to the scratchpad after allocating `numBytes` of storage. A null pointer is returned if there is not enough space.

The `vbx_sp_free()` function frees **all allocations** from the scratchpad. Use it like a master reset switch!

You cannot de-allocate individual calls to `vbx_sp_malloc()`. Instead, the current position of the scratchpad pointer can be saved onto a stack using the `vbx_sp_push()` function. The scratchpad pointer keeps track of the current high-water mark of scratchpad allocations. To restore the current position of the scratchpad pointer, use `vbx_sp_pop()`.

The recommended use of `vbx_sp_push()` and `vbx_sp_pop()` are at the beginning and end of a subroutine, respectively. This will deallocate all of the `vbx_sp_malloc()` calls made in the subroutine. The recommended way of 'returning a value' in the scratchpad is to have the caller reserve the space in advance.

The scratchpad pointer can be explicitly manipulated using the `vbx_sp_set()` and `vbx_sp_get()` functions. The purpose of these is for users to provide a more advanced scratchpad memory management feature set, e.g. similar to a traditional malloc.

## 3.4 Memory Management

The memory management APIs are:

```
vbx_shared_alloca( numBytes );

vbx_shared_malloc( numBytes );

vbx_shared_free( ptr );
```

The `vbx_shared_alloca()` call will allocate vector storage on the local call stack, similar to the gcc `alloca()` function. Upon return from the current function, all storage allocated with `alloca()` is automatically freed.

The `vbx_shared_malloc()` call will allocate vector storage on the heap, similar to the C library `malloc()` function. Each block of memory allocated with this function **must** be freed using `vbx_shared_free()`.

Internally, both of these memory allocation functions will reserve a chunk of memory that is slightly larger than what is requested in order to accommodate padding requirements of the DMA engine and to store memory management information. Also, the pointer returned by these functions is always marked *uncached* to avoid data consistency problems with the MXP vector engine.

If a large amount of scalar operations must be performed on this region, it may be worthwhile to obtain a *cached* pointer, and then flush the data cache before passing control to the MXP vector engine. To do this, use the `vbx_remap_cached()` and `vbx_remap_uncached()` operations.

## 3.5 Global Macros

The following global macros can be set to control the VBX API software library system:

- `VBX_SKIP_ALL_CHECKS` set to 1 for high-performance code with minimal safety or bounds checks
- `VBX_DEBUG_LEVEL` set to 0 for quiet, 1 or 2 for verbose debug printing

# 4 Vector Expressions

A Vector is a complex type defined in `Vector.hpp` that allows us to use simple operators to manipulate vectors. This provides an interface that is cleaner to read and easier to develop with.

## 4.1 Declaration

The standard constructor has the signature `template<typename T> Vector<T>(size_t length);` The Vector is allocated with `length` space on the scratchpad. This length is constant and cannot be changed after construction.

It is sometimes necessary to create a vector using space previously allocated on the scratchpad, for this reason there is an additional constructor: `template<typename T> Vector<T>(T* ptr,size_t length);`. That constructor creates a Vector using *length* elements starting at at the address *ptr* in the scratchpad.

## 4.2 Vector Expressions

A Vector expression is composed of operators linking combinations of Vectors, Enumerated types, and scalars. Using the operations detailed below, we can create expressions like `va=vb>>2 * VBX::ENUM` or `vc.cond_mov(va<ab,4-vd)`.

## 4.3 Operators

### 4.3.1 Arithmetic operators

The following element-wise operators are defined for Vectors:

| Operator | Description | Analogous instruction |
|---|---|---|
| `a+b` | Addition | `VADD` |
| `a-b` | Subtraction | `VSUB` |
| `absdiff(a,b)` | Absolute difference | `VABSDIFF` |
| `a*b` | Multiplication | `VMUL` |
| `mulfxp(a,b)` | Fixed point multiplication | `VMULFXP` |
| `mulhi(a,b)` | High bits of a multiply | `VMULHI` |
| `a^b` | Bit-wise exclusive or | `VXOR` |
| `a&b` | Bit-wise and | `VAND` |
| `a\|b` | Bit-wise or | `VOR` |
| `a<<b` | Shift right operator | `VSHL` |
| `a>>b` | Shift left operator | `VSHR` |
| `a<b` | Less than comparison | `VSUB`,`VCMV_LT` |
| `a>b` | Greater than comparison | `VSUB`,`VCMV_GT` |
| `a<=b` | Less than or equal comparison | `VSUB`,`VCMV_LEZ` |
| `a>=b` | Greater than or equal comparison | `VSUB`,`VCMV_GEZ` |

| Operator | Description | Analogous instruction |
|---|---|---|
| `a==b` | Equality comparison | `VSUB`,`VCMV_Z` |
| `a!=b` | Inequality comparison | `VSUB`,`VCMV_NZ` |

Because it is inefficient from a hardware standpoint,the following operations are not supported:

- Scalar shift by a vector: `5 >> va` or `5 << va`
- Vector shifted by an enum: `va >> ENUM` or `va << ENUM`
- A Vector subtracted from an enum: `ENUM - va`

#### 4.3.2 Logical Operators

In addition to the above arithmetic operator the logical operators `||` or `&&` and `!` are defined. Which allow combining comparison operators. `||` and `&&` usually take 3 instructions to to resolve, but `!` can usually be resolved without any additional instructions.

### 4.4 Type Conversion

When using one of the above operators care must be made to ensure that types of the 2 Vector operands match. For instance a `Vector<vbx_word_t>` cannot be added to a `Vector<vbx_uword_t>` without first casting it to one of the operands to the type of the other using the `.cast<TYPE>()` member function.

An Operation inherits the type of it operands, so a the expression `va+ENUM` would be a binary operator with the same type as `va`. Binary operators can also be cast using the same member function. so if `vb` is a `Vector<vbx_word_t>` and `vc` is a `Vector<vbx_half_t>` the following expression is valid `vb + (vc * 3).cast<vbx_word_t>()` but the expression `vb + (vc * 3)` is not valid because the type of `vb` does not match the type of the binary operation `vc * 3`.

It is important to note that a the destination of an expression (The Vector that the expression is being assigned to) does not have to match. So using the same vectors `vb` and `vc` the statement `vb = vc - 5` is valid.

### 4.5 Assignment

Vector expressions are evaluated when they are assigned to another Vector, we have two types of assignment; normal and conditional.

#### 4.5.1 Normal Assignment

Any Vector Expression can be assigned to a vector. The length of the Vector being assigned to is what determines the vector length of the calculations.

### 4.5.2 Conditional Assignment

A conditional assignment take the form `dest.cond_move(condition, iftrue)`.

Conditional assignment works mostly the same as regular assignment with the exception that for a given element `i`, the value `iftrue[i]` is only moved if `condition[i]` is true.

To check if the overflow or carry flag is set, it is possible to use the `fs()`,`fc()`,`overflow()`, and `carry()` member functions of the vector class. For instance saturating add could be implemented as follows: `va = vb+20; va.cond_move(va.overflow(),255);`

**Note about comparisons**

It is important to note that the comparison operators do not resolve to 1 or 0 like a regular c/c++ comparison, rather they resolve to the difference between a and b, an a state variable that remembers what type of comparison it was last used. This state variable is reset to `VCMV_NZ` if any other operation is done to that vector.

*For example:*

```
VBX::Vector<vbx_word_t> va(3),vb(3),vc(3);
va=VBX::ENUM;
vb=1;
vc=va<vb;
vb.cond_mov(vc,0);
//at this point vc will contain -1,0,1 and vb will contain 0,1,1
//because vc remembers that it is a less than relationship
```

on the other hand

```
VBX::Vector<vbx_word_t> va(3),vb(3),vc(3);
va=VBX::ENUM;
vb=1;
vc=va<vb;
vc+=0;   //<-- note this extra addtion
vb.cond_mov(vc,0);
//at this point vc will contain -1,0,1 and vb will contain 1,0,1
//because vc no longer remembers that it is a less than relationship,
//because it's last operation was an addition
```

## 4.6 Accumulating Vectors

To accumulate (sum) a vector or expression, use the function `accumulate(vexpr expr, size_t len)`. The result of that function can be assigned to a `VBX::accum<T>` where T is an integral type. An `VBX::accum<T>` is a integer that resides in the scratchpad memory. It can mostly be treated like any other integer, except that the fact that it can be written to by both the vector engine and the scalar engine presents some synchronization issues. Assigning a regular integer to an `accum<T>` is done in order with other vector operations, not with the scalar CPU instructions. There are two ways of reading the value: `async_getval()` and `sync_getval()`. The asynchronous method reads the current value in the memory without waiting for the MXP to finish operations. The synchronous method issues a `vbx_sync()` instruction before reading the value, which assures that all vector instructions have completed. For convenience there is also a implicit conversion to integer which does a synchronous read.

Using the array subscript operator with an integer rather than a range (`upto`) creates a `accum<T>` for that element. Using this feature it is possible to do an accumulate into an element of a Vector i.e.: `va[5]=accumulate(vb);`

## 4.7 Multi-Dimensional Operations

Two and three dimensional operations can be used to increase code density and provide higher performance in some cases.

To declare a 2D Vector use the constructor `Vector<T,2>(columns,rows,increment)`. Where `rows`and `columns` are the dimensions of the matrix and `increment` is the number of elements between two successive rows.

To declare a 3D Vector use the constructor `Vector<T,2>(columns,rows,increment2,matrices,increment3)`. Where `rows`and `columns` are the dimensions of the matrix, `increment2` is the number of elements between two successive rows, `matrices` is the number of matrices and `increment3` is the number of elements between successive matrices.

It is also possible to accumulate a vector into a lower order Vector and combine the result with a vector if that lower vector in the following manner `v1d + accumulate( v2d * 4)`.

Note that it is important that `v2d` has as many rows as `v1d` has elements so that the dimensions match. The library cannot check for this at compile time. So it is left to the programmer to make sure this happens.

Rather than declaring a vector as a 2D vector, it is possible to reshape a 1D vector into a 2D Vector using the `.to2D(int cols,int rows,ssize_t increment)` member function. This member functions creates a temporary Vector that aliases the original vector.

Using this multi dimensional API it is possible to implement a simple FIR filter in the following manner:

```
void fir(int* output,int* input,int* taps, int sample_size,int num_taps)
{
   Vector<int> v_out(sample_size-num_taps);
   Vector<int> v_in(sample_size);
   Vector<int,2> v_taps(num_taps,num_taps,sample_size,0);
   v_in.dma_read(input);
   v_taps.dma_read(taps);
   v_out=accumulate(v_in.to2D(num_taps,sample_size-num_taps,1)*v_taps);
   v_out.dma_write(output);
}
```

*Note* Neither masked instructions (detailed below) or Logical operators (&&,||) work with 2 or 3 dimensionalq operations.

## 4.8 Vector Slices

It is possible to work with only a portion of a vector by creating a Slice of a vector. This is done with the `[]` operator. `va[0 upto 15]` will create a vector that refers to the first 15 elements of `va`.

When dealing with 2D vectors it is possible to refer to a sub-matrix in the following manner `v2d[0 upto 15 , 0 upto 5]` this creates a vector of the first 15 elements of the first 5 rows.

No data is copied during any of these operations, it is just viewed differently.

## 4.9 Prefetching Data

When working with large data sets, it is necessary to only work with a portion of the data at a time. To get maximum performance it is often desirable to prefetch the next portion of data while working with the current data. This maximizes memory and vector instruction bandwidth. This is known as double buffering data.

To facilitate this, there exists a class: `Prefetcher<T>` with the constructor `Prefetcher<T>(int num,size_t vec_size,T* start,T* end,unsigned int vec_incr)` This constructs an object that lets you work with `num` vectors of `vec_size` elements each.

Using the `operator[] (int i)` allows you to access the ith vector.

The method `fetch()` starts a DMA transfer `vec_size` elements from `start+vec_incr * n` into the next available vector where n is the number of times you have called the `fetch()` method. If `start+vec_incr * (n+1) > end` then only `end - (start+vec_incr * n)` elements will be transfered, and the vector will be resized to that size as well. This makes the situation where the last dma transfer is smaller than the usual chunk size easier to handle. In the case where all of data betwee `start` and `end` has been transfered already, calling `fetch()` will simply rotate the internal vector list without doing any DMA so that the `[]` operator behaves as expected.

The following example takes an image and makes an output image where each row is the sum average of the row above and below it. (each row starts `pitch` words after the previous)

```
void image_average(vbx_word_t* out_img, vbx_word_t* in_img, int rows,int cols,int pit
{
    VBX::Prefetcher<vbx_word_t> in_rows(3,cols,in_img,in_img+rows*pitch,pitch);
    //fetch first three rows
    in_rows.fetch();
    in_rows.fetch();
    in_rows.fetch();
    VBX::Vector<vbx_word_t> result(cols);
    //skip first and last rows
    for(int row=1;row<(rows-1);row++){
        //the row being fetched on the next line will be used in
        //the next iteration.
        in_rows.fetch();
        //The subscript operator returns a reference to a vector,
        //subscript 0 refers the last Vector in the prefetcher fifo.
        //(the image row with a lower row number)
        //It will be overwritten on the next fetch().
        Vector<vbx_word_t>& rowA=in_rows[0];
        Vector<vbx_word_t>& rowB=in_rows[2];
        //note how rowA and rowB are declared as references,
        //otherwise there would be unnecessary data copy operations
        result=( rowA + RowB )>>1 /*shift is divide by two*/
        result.dma_write(out_image+row*pitch);
    }
}
```

## 4.10 Masked Blocks

Masked vector instructions are handled in a manner similar to if blocks in c. use the macro `Vector_mask( condition, length){...}`. The condition parameter is a vector expression that calculates which elements are masked, length is the vector length to use when creating the mask. It is possible to narrow the current mask with `Vector_mask_narrow( condition , length)`. Narrowing does *not* create another block.

As an example:

```
VBX::Vector<T> va(5);
va = VBX::ENUM; // <-- not masked
// va now contains {0,1,2,3,4}
```

```
Vector_mask(va < 2, 5){
   va = 10; //<-- masked calculation
// va now contains {10,10,2,3,4}
}
va +=1; //<-- not masked calculation
// va now contains {11,11,3,4,5}
```

Note that nested masked blocks are **not** supported, so the following is undefined.

```
//BAD EXAMPLE, IS NOT SUPPORTED!
Vector_mask(va < 2, 5){
   Vector_mask(vb < 2, 5){
   ...
   }
}
```

You can however do :

```
Vector_mask(va < 2 && vb < 2 , 5){
   ...
}
```

or

```
Vector_mask(va < 2  , 5){
   Vector_mask_narrow(vb < 2, 5){
   ...
   }
}
```

When doing small amounts of masked calculations (i.e. 3 or less instructions). It is usually more efficient to use a conditional move rather than set up the mask.

## 4.11 Low Level Operations

Sometimes it is desirable to access MXP functionality that is not *yet* available via the high level Vector class. For instance 2D and 3D operations are not supported. For that reason we have striven to make it as easy as possible to mix `vbxx()` calls with Vector expressions. If it is necessary to get access to the scratchpad pointer simply use the `.data` member of the Vector class. Or, if you would like to use a vector object in `vbxx()` calls directly, that is supported as well. For example:

```
VBX::Vector<vbx_word_t> va(5);
//these next two lines are equivalent
vbxx(VADD,va,1,va);
vbxx(VADD,va.data,1,va.data)
```

## 4.12 Best Practice

Group together arithmetic involving like types ie:

```
Vector<vbx_word_t> vd(size),va(size);
Vector<vbx_half_t> vb(size),vc(size);
//This is two operations VSW, VSW
vd=va+5 + 1;
//This is one operation VSW
vd=va+(5+1);
//this is 4 operations VSHW(type conversion),VSWW(addition),VSHW(type conversion),VSW
vd=va+vb+vc;
//this is two operations VSWH(type conversion and addition),VSWW(addition)
vd=va+(vb+vc);
```

## 4.13 Gotchas

This library has not been exhaustively tested so there could be bugs in it.

Watch out for failed scratchpad allocation. It is possible that you will get a null pointer for your scratchpad allocation which can be interpreted by the MXP as start of scratchpad destroying data.

When dropping down to low level `vbxx(...)` calls inside of a masked block, the masked or unmasked instructions must be explicit. The API does not automatically convert `vbxx(...)` to `vbxx_masked(...)`.

One of properties of templates is that sometimes compile time errors are not uncovered until a template is used. So if you use the library in a way that I intended it to be used, but have never actually used, you run the risk of having the compiler spit out a long list of errors. Please let me (joel@vectorblox.com) know if this happens to you.*

# 5 Instruction and DMA Hazards

The VectorBlox MXP architecture allows the DMA engine, the MXP vector engine, and the host processor to run concurrently.

In particular, vector instructions and DMA operations are deposited into a common DMA/instruction queue, returning control immediately back to the host processor before those operations execute. This increases performance through parallelism, for example it provides the opportunity to hide all memory latency through double-buffering with the DMA engine. However, such concurrency can sometimes be difficult for a programmer to manage.

To minimize the occurrence of hazards, the VectorBlox MXP architecture handles most data race conditions automatically in hardware, thus relieving a significant burden from the software programmer.

The list of hazards are:

1. hazards between vector instructions **automatic**
2. hazards between DMA operations and vector instructions **automatic**
3. hazards within a single instruction **manual**
4. hazards with the host processor **manual**

Of these, the first two hazards are handled automatically by the VectorBlox MXP hardware. The last two hazards require manual programmer care and intervention.

## 5.1 Hazards between Vector Instructions

Hazards between vector instructions are automatically handled by inserting a bubble into the vector pipeline. The bubble allows previously issued instructions to make forward progress while the current instruction is stalled until the hazards is resolved.

A hazard occurs when a new instruction reads from the destination of an earlier instruction, and that earlier instruction is still in the pipeline. While some processors apply data forwarding to improve performance, the overhead of data multiplexing is significant. Hence, the vector engine inserts a bubble instead. Thus, for maximum performance, programmers should avoid reading immediately after writing to the same scratchpad address; instead, another instruction may be inserted in between the two dependent instructions. This is not a problem if using very long vectors because the earlier instruction has likely already written back its result to the vector starting address before the dependent instruction issues.

Since all vector instructions execute in program-order, there are no write-after-write or write-after-read hazards between instructions.

## 5.2 Hazards between DMA Operations and Vector Instructions

Hazards between DMA operations and vector instructions are handled automatically by inserting a bubble into the pipeline when a hazard is detected. A pipeline bubble prevents new operations or instructions from issuing, but allows previously-issued operations or instructions to make forward progress.

A hazard is defined as:

- instruction reading from the destination of a DMA transfer `vbx_dma_to_vector()`
- instruction writing to the source of a DMA transfer `vbx_dma_to_host()`

- `vbx_dma_to_host()` DMA reading from the destination of the vector instruction
- `vbx_dma_to_vector()` DMA writing to the source of the vector instruction

When a hazard occurs, a subsequent DMA operation or vector instruction is stalled until the conflict has been resolved. In the case of conflicting with a DMA operation, it is resolved as soon as the DMA proceeds past the region of overlap (i.e., the DMA operation does not need to finish completely).

## 5.3 Hazards within a Single Instruction

Hazards may occur within a single instruction. Since these are not automatically detected, they require some care by the programmer.

Hazards occur when the source and destination vectors overlap in a particular way. In particular, there is no hazard in the following common cases:

1. `dest` overlaps perfectly with `srcA`, and overlaps perfectly with `srcB`
2. `dest` overlaps perfectly with `srcA`, but does not overlap with `srcB`
3. `dest` overlaps perfectly with `srcB`, but does not overlap with `srcA`
4. `dest` overlaps with `srcA`, and overlaps perfectly with `srcB`, and `dest` $<$ `srcA`
5. `dest` overlaps with `srcB`, and overlaps perfectly with `srcA`, and `dest` $<$ `srcB`
6. `dest` overlaps `srcA`, but does not overlap with `srcB`, and `dest` $<$ `srcA`
7. `dest` overlaps `srcB`, but does not overlap with `srcA`, and `dest` $<$ `srcB`

By 'overlaps perfectly', we mean the pointers are equal. In case 1, the three operands are the same pointer (`dest` = `srcA` = `srcB`). In cases 2 and 3, the destination perfectly overlaps with just one source, and there is no overlap with the other. These three cases of perfect overlap do not present any hazards.

In cases 4 through 7, the overlap represents a copy-backward operation, where the source address is always higher than the destination address in the overlap region. These copy-backwards operations do not present any hazards.

The remaining (unlisted) cases of overlap represent a copy-forward operation, where the destination address is higher than one of the source addresses. This is a dangerous operation, because the address may be written before it is read. Hence, a read-after-write hazard occurs.

A copy-forward operation must be used with great care. For a small copy-forward distance, there are two potential issues. The first issue is that some programmers may rely upon the pipeline depth to delay the writeback until after the scratchpad address has been read. This only works for a copy-forward distance that is shorter than the pipeline depth; since this relies upon implementation details of the processor which may change without notice, we do not encourage this type of programming. However, even if the copy forward distance is small enough, another issue may arise. If an earlier DMA operation is in progress, and causing the copy-forward to stall, then the copy-forward will probably fail. To avoid this problem, programmers must use `vbx_sync()` before any operation with a small copy-forward distance.

For a large copy-forward distance, we recommend using a 2D vector operation that iterates through the vector backwards, where the vector length is set to the scratchpad width.

A VBXware function, `vbx_vec_move()` handles the cases of overlap relating to a vector move operation (`VMOV`).

Note that 2D and 3D matrix operations have the same problems caused by overlap between rows. Again, copy-backward operations do not pose any difficulty. However, copy-forward operations should not be relied upon. If copy-forward operations must be done, the distance must be small and a `vbx_sync()` operation must be performed beforehand.

## 5.4 Hazards with the Host Processor

The host processor can read or write individual entries in the scratchpad through the scratchpad pointers. However, doing this while the vector engine is executing, or while the DMA engine is executing, may present a race condition.

To eliminate race conditions, the programmer should call `vbx_sync();` before accessing any scratchpad entries. This ensures that all previously issued DMA operations and vector instructions have completed before the access.

In some cases, it may be desirable to simply poll the scratchpad, e.g. to monitor progress without stalling. In this case, a synchronization call is not necessary. However, there are no guarantees whether prior operations have been fully or even partially completed.

## 5.5 Overlapping Communication with Computation

In order to get maximum performance it is necessary to overlap communication with computation. When processing large data sets, this is usually done by breaking the computation into chunks depending on the size of the scratchpad, and processing one chunk while transferring in the next. This allows the time spent transferring data to and from the scratchpad to be hidden during the actual computation.

For instance, consider the following code, which computes the cube of *a* and puts the result in *b*. It uses chunks of size *M* and has a total data set of size *M*N*.

```
vbx_word_t *v_a = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_b = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_set_vl( M );
for( i = 0; i < M*N; i += M ){
    vbx_dma_to_vector( v_a, a+i, M*sizeof(vbx_word_t) );
    vbx( VVW, VMUL, v_b, v_a, v_a );
    vbx( VVW, VMUL, v_b, v_b, v_a );
    vbx_dma_to_host( b+i, v_b, M*sizeof(vbx_word_t) );
}
```

If we want better performance, we need to rewrite the code to overlap computation with communication. One common way of doing this is double buffering. In double buffering, one buffer will hold data currently being processed, while the other will be used for transferring data to and from memory.

```
vbx_word_t *v_a0 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_a1 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_b0 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_b1 = vbx_sp_malloc( M*sizeof(vbx_word_t) );
vbx_word_t *v_tmp;

vbx_set_vl( M );
vbx_dma_to_vector( v_a0, a, M*sizeof(vbx_word_t) );
for( i = 0; i < M*N; i += M ){
    if( i < M*N-M ){
        vbx_dma_to_vector( v_a1, a+i+M, M*sizeof(vbx_word_t) );
    }
    vbx( VVW, VMUL, v_b0, v_a0, v_a0 );
    vbx( VVW, VMUL, v_b0, v_b0, v_a0 );
    vbx_dma_to_host( b+i, v_b0, M*sizeof(vbx_word_t) );

    //Swap buffers by changing pointers
    v_tmp = v_a0; v_a0 = v_a1; v_a1 = v_tmp;
```

```
        v_tmp = v_b0; v_b0 = v_b1; v_b1 = v_tmp;
    }
```

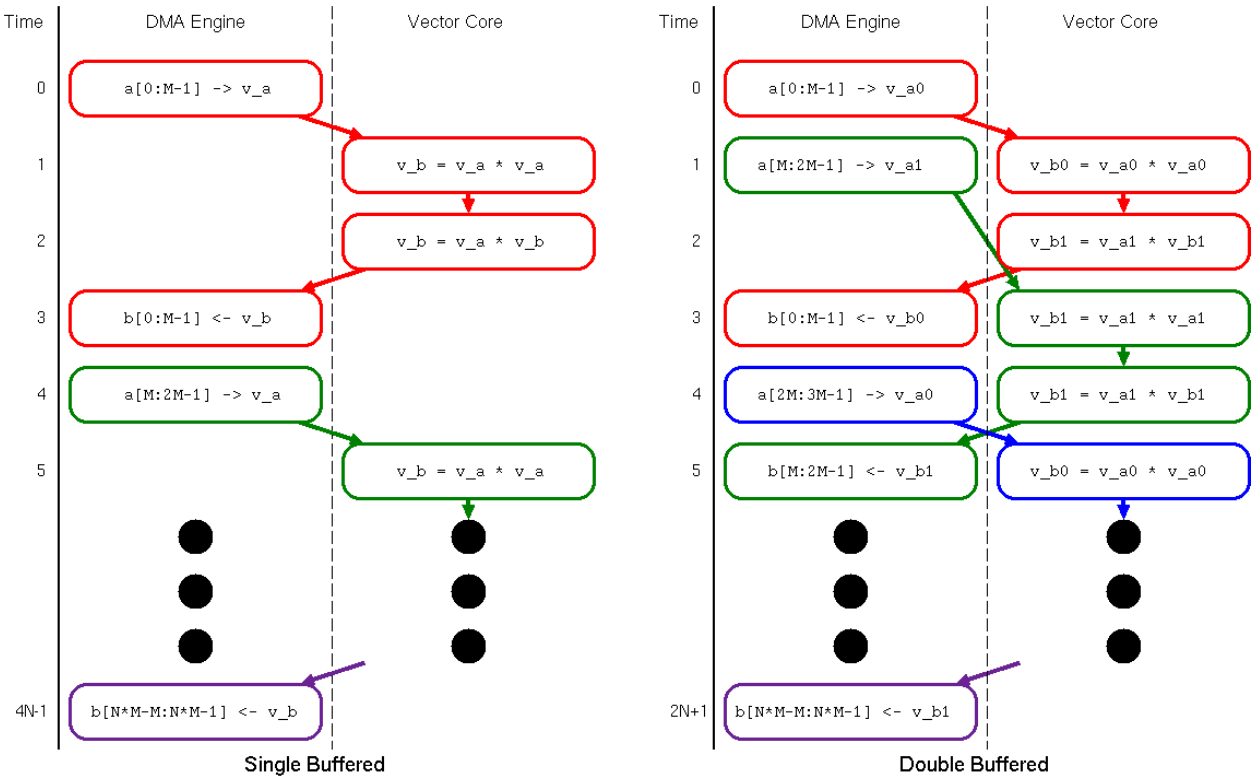


Figure 1: Communication and computation over time with no overlap

# 6 Runtime Checks

The VBX API supports checking parameters given to the api calls. The checks are enabled at compile time, using the following defines:

```
VBX_RTC_ALL
VBX_RTC_NONE
VBX_RTC_SP_BOUNDS
VBX_RTC_COP_FWD
VBX_RTC_DMA
VBX_RTC_VEC_LEN
```

The first macros , when defined, overide the other four, but cannot be both defined.

- `VBX_RTC_SP_BOUNDS` when calling vbx*() warn if vectors extend outside of scratchpad
- `VBX_RTC_COP_FWD` when calling vbx*() warn if there exists a copy forward hazard
- `VBX_RTC_DMA` when calling vbx_dma*() warn if host buffer overlaps without the scratchpad or the vector buffer extends outside of the scratchpad
- `VBX_RTC_VEC_LEN` When calling `vbx_set_vl()`, `vbx_set_2D()`, or `vbx_set_3D()` make sure the sizes don't exceed the size of the scratchpad

If it is your intent to violate one of these warnings, and you want to suppress them for brief periods, you can suppress/express them at runtime. Using `SUPPRESS_RT_CHECK( check )` or `EXPRESS_RT_CHECK( check )` respectively, where `check` is one of `RT_CHECK_SP_BOUND` , `RT_CHECK_COP_FWD`, `RT_CHECK_DMA`, `RT_CHECK_VEC_LEN`, or `RT_CHECK_ALL`.

# 7 VBX Portability Library

Vectorblox provides a selection of functions defined in vbx_port.h to provide portability between the ARM cortex-A9, Nios II and Microblaze processors. The functions provided fall into two categories; Timing and Cache Management

## 7.1 Timing

The Timing functions are as follows:

```
int vbx_timestamp_start();

unsigned vbx_timestamp_freq();

vbx_timestamp_t vbx_timestamp();
```

Using These functions it is possible to calculate the execution time of code.

`vbx_timestamp_start()` Resets the counter to zero and starts the timer running.

`vbx_timestamp_freq()` Returns the frequency of the timer tick in Hz

`vbx_timestamp()` Returns the current value of timer.

### 7.1.1 Timestamp Example

```
#include <stdio.h>
#include "vbx_port.h"

int main(void)
{
    // Declarations
    vbx_timestamp_t start,end;
    const float ms_per_sec = 1000.;
    unsigned freq;
    float milliseconds;

    // make the timer start ticking
    vbx_timestamp_start();
    // get the timer frequency
    freq = vbx_timestamp_freq();
    // save the time at the beginning of calculations
    start = vbx_timestamp();

    // do some work

    // save the time at the end of calculations
    end = vbx_timestamp();
    // calculate elapsed milliseconds
    milliseconds = (float)(end-start) / freq * ms_per_sec;
    // Print the results
    printf("Took %u timer ticks -> %f ms\n" , end-start, milliseconds);
    return 0;
}
```

## 7.2 Cache Management

The cache management functions are as follows:

```
volatile void* vbx_uncached_malloc(unsigned size);

volatile void* vbx_uncached_alloca(unsigned size);

void vbx_uncached_free(volatile void* ptr);

void vbx_dcache_flush_all();

void vbx_dcache_flush_line(void* ptr);

void vbx_dcache_flush(void* ptr,int len);

void* vbx_remap_cached(volatile void *ptr,unsigned len);

volatile void* vbx_remap_uncached(void* ptr);

volatile void* vbx_remap_uncached_flush(void* ptr,unsigned len);
```

The MXP cannot directly read the CPU cache, so it must read and write to main memory instead. This means that the programmer must used uncached memory, or flush the data from the cache before doing DMA or reading from the scratchpad. Use `vbx_uncached_malloc()` or `vbx_uncached_alloca()` in place of `malloc()` and `alloca` respectively to get pointers to uncached buffers. Use the `vbx_flush_*()` to flush the cache. `vbx_remap_cached(ptr)` returns an uncached pointer to the same data as `ptr`. To get an uncached pointer and flush the cache at the same time, use ‘vbx_uncached_flush();

Note That in order to use either the timestamp functions or the cache management functions, you have to include `vbx_port.h`

# 8 VBXware Library

The VBXware library is an example library showing advanced use of the VBX API .The VBXware library can be used for illustrative purposes, or used directly in user applications.

## 8.1 VBXware Functionality

The VBXware library provides the following functionality:

- Fixed Point Math
- Finite Impulse Response (Matrix and Vector)
- Median Filter
- Matrix Multiply
- Motion Estimation
- Sobel Edge Detection
- Matrix Transpose
- RGB To Luma Conversion
- Vector Addition
- Vector Copy
- Vector Exponentiation
- Vector Reversal

For further information on this the VBXware library, see the documentation.

## 8.2 VBXware Templates

Many of the functions in the VBXware libraries use the template engine. The VBX temperate engine makes it easy to create functions for all six vector types: word, half, byte, uword, uhalf and ubyte.

### 8.2.1 Summary

In a nutshell the templating engine does the following:

```
VBX_T(vbw_vec_add)(); -> vbx_vec_add_word();
                      -> vbx_vec_add_half();
                      -> vbx_vec_add_byte();
                      -> vbx_vec_add_uword();
                      -> vbx_vec_add_uhalf();
                      -> vbx_vec_add_ubyte();
```

```
VV(T) -> VVW
      -> VVH
      -> VVB
      -> VVWU
      -> VVHU
      -> VVBU
```

```
SV(T) -> SVW
      -> SVH
      -> SVB
      -> SVWU
      -> SVHU
      -> SVBU
```

And so on. . .

After including the relevant files all of the vbx_vec_add_*() functions are defined, and can therefore be called.

### 8.2.2 Explanation

The mechanics of how to set up your code to use this is somewhat complicated. To explain further how this works, let us look for a moment at the addition example located in the VBXware directory in the files vbw_add_t.* and vbw_add_all.*

vbw_vec_add_all.c:

```
#define VBX_TEMPLATE_T VBX_BYTESIZE_DEF
#include "vbw_vec_add_t.c"

#undef VBX_TEMPLATE_T
#define VBX_TEMPLATE_T VBX_HALFSIZE_DEF
#include "vbw_vec_add_t.c"

#undef VBX_TEMPLATE_T
#define VBX_TEMPLATE_T VBX_WORDSIZE_DEF
#include "vbw_vec_add_t.c"

#undef VBX_TEMPLATE_T
#define VBX_TEMPLATE_T VBX_UBYTESIZE_DEF
#include "vbw_vec_add_t.c"

#undef VBX_TEMPLATE_T
#define VBX_TEMPLATE_T VBX_UHALFSIZE_DEF
#include "vbw_vec_add_t.c"

#undef VBX_TEMPLATE_T
#define VBX_TEMPLATE_T VBX_UWORDSIZE_DEF
#include "vbw_vec_add_t.c"
```

vbw_vec_add_t.c:

```
#include "vbw_vec_add_t.h"
void VBX_T(vbw_vec_add)(vbx_sp_t *v_out, vbx_sp_t *v_in1, vbx_sp_t *v_in2)
{
```

```
    vbx(VV(T), VADD, v_out, v_in1, v_in2);
    }
```

vbw_vec_add_all.h:

```
    #define VBX_TEMPLATE_T VBX_BYTESIZE_DEF
    #include "vbw_vec_add_t.h"

    #undef VBX_TEMPLATE_T
    #define VBX_TEMPLATE_T VBX_HALFSIZE_DEF
    #include "vbw_vec_add_t.h"

    #undef VBX_TEMPLATE_T
    #define VBX_TEMPLATE_T VBX_WORDSIZE_DEF
    #include "vbw_vec_add_t.h"

    #undef VBX_TEMPLATE_T
    #define VBX_TEMPLATE_T VBX_UBYTESIZE_DEF
    #include "vbw_vec_add_t.h"

    #undef VBX_TEMPLATE_T
    #define VBX_TEMPLATE_T VBX_UHALFSIZE_DEF
    #include "vbw_vec_add_t.h"

    #undef VBX_TEMPLATE_T
    #define VBX_TEMPLATE_T VBX_UWORDSIZE_DEF
    #include "vbw_vec_add_t.h"
```

vbw_vec_add_t.h:

```
    #include "vbw_template_t.h"

    void VBX_T(vbw_vec_add)(vbx_sp_t *v_out, vbx_sp_t *v_in1, vbx_sp_t *v_in2);
```

What happens here is the function name is; vbx_vec_add_all.h includes vbx_vec_add_t.h multiple times, each time with `VBX_TEMPLATE_T` defined to a different type. In vbx_vec_add_t.h the macro expands to `vbw_vec_add_word` if `VBX_TEMPLATE_T` is defined as `VBX_WORDSIZE_DEF` etc. vbw_template_t.h redefines `vbx_sp_t` to the correct type based on the `VBX_TEMPLATE_T` definition as well. One last thing to note is in vbx_vec_add_t.c the mode parameter of the `vbx()` call is `VV(T)` this will be translated to the correct mode(`VVW` when `VBX_TEMPLATE_T` equals `VBX_WORDSIZE_DEF`. )

# 9 Simulator

The VectorBlox simulator is a x86 library for Linux or windows (via mingw) that accurately simulates the VBX MXP. The following sections document the functionality of the simulator.

It also provides the ability to count instructions, a provide a rough estimate of cycle counts.

## 9.1 Usage

The simulator provides the same functionality as the regular MXP. It works in place of the BSP. Simply linking with libvbxsim.a rather than libbsp.a or libxil.a will allow using VectorBlox functions without hardware to run on.

## 9.2 Initialization

To initialize the simulator the following function is provided.

```
void vbxsim_init( int num_lanes,
                  int vci_lanes,
                  int scratchpad_capacity_kb ,
                  int max_masked_waves,
                  int fxp_word_frac_bits,
                  int fxp_half_frac_bits,
                  int fxp_byte_frac_bits);
```

Using this initialization function it is possible to create the desirable configuration. The parameters are analogous to the corresponding parameter in the hardware configuration.There is a symmetrical function `vbxsim_destroy()` that cleans up the simulator, freeing any memory.

## 9.3 Diagnostics

The simulator provides the following functions to query diagnostics, and modify it's behaviour.

```
#define MAX_DMA_ALIGN 128
#define MAX_VEC_LANE  10
struct simulator_statistics{
    union{
        struct {
            unsigned VMOV[MAX_VEC_LANE];
            unsigned VAND[MAX_VEC_LANE];
            unsigned VOR[MAX_VEC_LANE];
            unsigned VXOR[MAX_VEC_LANE];
            unsigned VADD[MAX_VEC_LANE];
            unsigned VSUB[MAX_VEC_LANE];
            unsigned VADDC[MAX_VEC_LANE];
            unsigned VSUBB[MAX_VEC_LANE];
            unsigned VMUL[MAX_VEC_LANE];
            unsigned VMULHI[MAX_VEC_LANE];
            unsigned VMULFXP[MAX_VEC_LANE];
            unsigned VSHL[MAX_VEC_LANE];
            unsigned VSHR[MAX_VEC_LANE];
            unsigned VROTL[MAX_VEC_LANE];
            unsigned VROTR[MAX_VEC_LANE];
```

```
            unsigned VCMV_LEZ[MAX_VEC_LANE];
            unsigned VCMV_GTZ[MAX_VEC_LANE];
            unsigned VCMV_LTZ[MAX_VEC_LANE];
            unsigned VCMV_GEZ[MAX_VEC_LANE];
            unsigned VCMV_Z[MAX_VEC_LANE];
            unsigned VCMV_NZ[MAX_VEC_LANE];
            unsigned VABSDIFF[MAX_VEC_LANE];
            unsigned VCUSTOM0[MAX_VEC_LANE];
            unsigned VCUSTOM1[MAX_VEC_LANE];
            unsigned VCUSTOM2[MAX_VEC_LANE];
            unsigned VCUSTOM3[MAX_VEC_LANE];
            unsigned VCUSTOM4[MAX_VEC_LANE];
            unsigned VCUSTOM5[MAX_VEC_LANE];
            unsigned VCUSTOM6[MAX_VEC_LANE];
            unsigned VCUSTOM7[MAX_VEC_LANE];
            unsigned VCUSTOM8[MAX_VEC_LANE];
            unsigned VCUSTOM9[MAX_VEC_LANE];
            unsigned VCUSTOM10[MAX_VEC_LANE];
            unsigned VCUSTOM11[MAX_VEC_LANE];
            unsigned VCUSTOM12[MAX_VEC_LANE];
            unsigned VCUSTOM13[MAX_VEC_LANE];
            unsigned VCUSTOM14[MAX_VEC_LANE];
            unsigned VCUSTOM15[MAX_VEC_LANE];
        }as_name;
        unsigned as_array[MAX_INSTR_VAL+1][MAX_VEC_LANE];
    }instruction_cycles;
    unsigned int instruction_count[MAX_INSTR_VAL+1];
    unsigned int set_vl;
    unsigned int set_2D;
    unsigned int set_3D;
    unsigned int dma_bytes;
    unsigned int dma_calls;
    unsigned int dma_cycles[MAX_DMA_ALIGN];
};
struct simulator_statistics vbxsim_get_stats();
enum dma_type {DEFERRED=0,  IMMEDIATE=1};
void vbxsim_set_dma_type(enum dma_type);
void vbxsim_reset_stats();
void vbxsim_print_stats()
void vbxsim_print_stats_extended();
void vbxsim_disable_warnings();
void vbxsim_enable_warnings();
```

The simulator keeps track of how many cycles of execution the MXP must perform for the calculations. It does this simultaneously for any number of lanes, if the number of lanes is $2^i$, $i \in [0, 10)$. To get examine these statistics use the function `vbxsim_get_stats()`, it returns the simulator_statistics structure that is described above. The counts can be reset back to zero using `reset_cycle_counts()`. Lastly,there is a pair of useful functions that print out the structure in a pretty format: `vbxsim_print_stats()` and `vbxsim_print_stats_extended()`.

The simulator has a limitation in that it is run synchronously with host, which means that race conditions that make bugs on the hardware would not show up in the simulator. To deal with this limitation we have developed two modes of DMA, `DEFERRED` and `IMMEDIATE`. When the simulator is in deferred mode (default), DMA requests are not done until a) vbx_sync() is called, b) a vbx() call depends on the data in the DMA request's buffers, or c) a vbx() call depends on data in a subsequent DMA request's buffer. When the simulator is in immediate mode, the simulator does all DMA synchronously. So it is as if there is a `vbx_sync()` call right after every DMA call. If an algorithm works for both modes on the simulator it should work on the hardware, the opposite is not necessarily true. To switch between these two modes, use `set_dma_type(enum dma_type)`.

## 9.4 Adding Custom Instructions

It is possible to implement custom instructions in Vectorblox's Matrrix processor, if your project makes use of this functionality, you may also want to simulate it as well.

To do this, the simulator provides a hook :

```
typedef void (*custom_instr_func)(vbxsim_custom_instr_t*);
void vbxsim_set_custom_instruction(int opcode_start,
                                   int internal_functions,
                                   int lanes,
                                   custom_instr_func fun);
```

where num is the custom instruction number(0 to 15), from zero to fifteen and func is a pointer to a function that will carry out the operations to emulate the hardware. The function takes one structure as a parameter. The following describes said structure (taken from vbx_sim.h) :

```
typedef struct{
        //bool reset;          ///< Global (hard) synchronous reset
        uint16_t valid;        ///< Current wavefront contains valid data
        char vector_start;     ///< First cycle of vector operation
        char vector_end;       ///< last cycle of vector operation
        void* dest_addr_in;    ///< Destination (writeback) address from address gener
        void* dest_addr_out;   ///< Destination (writeback) address to be written  (OU
        char sign;             ///< Signed operation
        int opsize;            ///< Datasize (00=Byte, 01=Halfword, 10=Word)
        void* byte_valid;      ///< Bytes containing valid data
        void* byte_enable;     ///< Bytes to be written to scratchpad              (OU
        void* data_a;          ///< Source A input data
        void* flag_a;          ///< Source A input flags
        void* data_b;          ///< Source B input data
        void* flag_b;          ///< Source B input flags
        void* data_out;        ///< Destination (writeback) data                   (OU
        void* flag_out;        ///< Destination (writeback) flags                  (OU
}vbxsim_custom_instr_t;
```