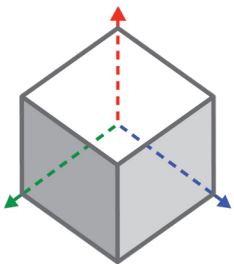

VectorBlox
embedded supercomputing

VectorBlox MXP Custom Instruction Manual

March 01, 2016

VectorBlox

http://vectorblox.com

Contents

1 Custom Instruction Basics

NOTE: MXP Custom Instructions are still experimental and subject to change.

MXP supports adding additional functionality through the use of *custom instructions*. From a software point of view, custom instructions execute the same way as basic vector instructions, using the `VCUSTOM0` to `VCUSTOM15` opcodes. Multiple basic vector instructions can be replaced by a single custom instruction if they share the same data inputs and output. Examples include bitwise operators (e.g. count leading zeros), different data types (floating-point, packed RGBA), and nonlinear functions (transcendentals). Some example custom instructions can be found in Example Custom Instructions.

1.1 Multiple Custom Instructions

MXP supports sixteen custom opcodes, allowing up to sixteen separate custom instructions to be connected at a time. The software programmer selects between the custom instructions using the sixteen different opcodes available (`VCUSTOM0` to `VCUSTOM15`).

Each VCI *port* has one or more functions each of which gets an opcode. The custom instruction hardware on a VCI port with N functions sees a one-hot N-bit `vci_valid` signal corresponding to which custom instruction opcode was dispatched (see HDL Signals to/from Custom Instruction for more information). The purpose of having a VCI port with multiple functions is to allow sharing of logic between functions. For instance, a divide circuit might be able to do both divide and modulo with very little modification, and therefore it would make sense to use two opcodes to select which function to perform. The starting opcode for each VCI port can be mapped arbitrarily into the sixteen opcode space, with additional opcodes mapped sequentially. For instance, if the divide/modulo VCI (with two functions) had its starting opcode set to `VCUSTOM5`, executing `VCUSTOM5` would use the VCI and set `vci_valid(0)` high when valid data existed, while executing `VCUSTOM6` would use the VCI and set `vci_valid(1)` high.

1.2 Custom Instruction Lanes

Custom instructions can range from very simple bitwise operators to complex operators such as divide or square root. Since a complex operator can be very large in area (as large as an entire MXP lane in the case of a fixed-point square root operator) it may be desirable to have fewer custom instruction lanes than vector lanes. MXP supports a separate `VCI_X_LANES` parameter for each custom instruction port that sets the number of custom instruction lanes. The number of custom instruction lanes must be fewer than or equal to the number of vector lanes, but does not need to be a power of two. If there are fewer custom instruction lanes than vector lanes, custom instructions will take correspondingly longer to process.

1.3 Modifying the Destination Address

In some cases a custom instruction may wish to modify the destination address where data will be written back. An example is a compress operation, where the vector length is reduced by removing some elements (such as those with a corresponding flag set). In this case the `vci_dest_addr_in` and `vci_dest_addr_out` signals need to be added to the VCI and pipelined the same way as the data/flag/byteenable signals. Instructions that don't modify the destination address don't need to include these signals. VCIs that modify the destination address cause a pipeline flush afterwards in order to avoid any potential hazards, so only instructions that need to modify the destination address should include the `vci_dest_addr_in` and `vci_dest_addr_out` signals.

1.4 Pipelining Custom Instructions

Custom instructions execute in parallel to the basic MXP execution pipeline. The number of pipeline stages used by each function of the VCI needs to be passed to the top level `VCUSTOMX_DEPTH` parameter of MXP for each opcode/function used by the VCI. If this parameter is not set correctly errors such as data being written back at the wrong address may occur. For VCIs with short pipelines, MXP adds extra internal registers to match the length of its execution pipeline. VCIs with deep pipelines (currently greater than two stages on Altera devices and four stages on Xilinx devices) are supported but have a small performance penalty as they currently force a pipeline flush. Other than the performance difference there is no visible difference to the software programmer or VCI designer.

1.5 Stalling and Stateful VCIs

Because of hazards within MXP, valid data may not be present every cycle. In this case the `vci_valid` signal will not be asserted during the stalled cycles. (see HDL Signals to/from Custom Instruction for more information). A pipelined VCI should produced valid data `VCUSTOMX_DEPTH` cycles after each `vci_valid` signal. For instance, if `VCUSTOMX_DEPTH` is 3 and `vci_valid` is asserted on cycles 0, 1, 3, and 4 then the VCI should produce data and byte enables on cycles 3, 4, 6, and 7.

A *stateless* VCI, where the output only depends on the inputs during one cycle, will have this behavior simply by implementing a free-running pipeline. A *stateful* VCI, where the output is dependent on the inputs over multiple cycles, must be set up with the same behavior. For instance, consider a VCI that performs a 3x3 2D convolution. This can be implemented using 2D instructions to input a new row every cycle; the first cycle would have one wavefront from row 0 as input, second cycle would be a wavefront from row 1, and so on. To deal with stalls in a stateful VCI, it is necessary to latch in new inputs when `vci_valid` is asserted yet keep the processing pipeline free running. Wrong behaviour will result if latching the input is counted as part of the pipeline. For instance, assume the 3x3 convolution took 5 cycles to compute. It might seem straightforward to set `VCUSTOMX_DEPTH` to 8 and have 3 stages of pipeline for latching in inputs followed by 5 cycles of processing. While this will work when data is presented every cycle, after a stall the 3 input stages may no longer contain valid data. The correct implementation sets `VCUSTOMX_DEPTH` to 5 and latches the inputs into a shift register only when `vci_valid` is asserted. This keeps the processing pipeline free-running, while keeping the input data from previous cycles valid.

There is a *warm up* period when the input shift register will not be full of valid data. In this example, until the third row has been input the input shift register contains some invalid data. This can be dealt with by modifying the destination address to not write over the first two rows of output or by just not using the first two rows in subsequent computations. Similarly, at the end of the vector (after the `vci_vector_end` signal is asserted) there is a *cool down* period where no more valid data is read in and the valid data in the pipeline working its way through (flushing).

1.6 Accumulated Custom Instructions

The reduction-accumulate used in *accumulated vector instructions* happens after the three stage custom instruction pipeline, so custom instructions can be freely mixed with reduction-accumulation. All writeback data with asserted byteenables will be summed by an accumulated custom instruction, and only a single value written back. The writeback only occurs on the last cycle of the vector instruction (or the last cycle of each row for 2D/3D vector instructions). In order to work with MXP's reduction-accumulation, custom instructions should not modify the destination address (see Modifying the Destination Address).

2 HDL Signals to/from Custom Instruction

Each custom instruction has the same set of signals consisting of control inputs and data input and output. Some signals are dependent on the number of custom instruction lanes (`VCI_X_LANES`) and the number of functions this VCI supports (`VCI_X_FUNCTIONS`).

Signal Name	In/Out	Bitwidth	Description
vci_clk	In	1	Global clock signal
vci_reset	In	1	Global (hard) synchronous reset
Control Signals			
vci_valid	In	VCI_X_FUNCTIONS	Current wavefront valid for function N (one-hot)
vci_vector_start	In	1	First cycle of vector operation
vci_vector_end	In	1	Last cycle of vector operation
vci_dest_addr_in	In	32	(optional) Destination (writeback) address from address generation
vci_dest_addr_out	Out	32	(optional) Destination (writeback) address to be written
Instruction Modifiers			
vci_signed	In	1	Signed operation
vci_opsize	In	2	Datasize (00=Byte, 01=Halfword, 10=Word)
Data Signals			
vci_byte_valid	In	VCI_LANES*4	Bytes containing valid data
vci_byteenable	Out	VCI_LANES*4	Bytes to be written to scratchpad
vci_data_a	In	VCI_LANES*32	Source A input data
vci_flag_a	In	VCI_LANES*4	Source A input flags
vci_data_b	In	VCI_LANES*32	Source B input data
vci_flag_b	In	VCI_LANES*4	Source B input flags
vci_data_out	Out	VCI_LANES*32	Destination (writeback) data
vci_flag_out	Out	VCI_LANES*4	Destination (writeback) flags

The `vci_clk` signal is used to clock the entire custom instruction. Since the MXP execution back-end is stall-free, there is no clock enable. `vci_reset` is a synchronous global reset and should clear all state within the custom instruction when asserted.

The *Control Signals* come from MXP's address generation logic. The `vci_valid` signal indicates there is valid input data on the current cycle, and is one-hot encoded to indicate which function has been selected. Custom instructions will not be issued every cycle, and even when issuing there may be pipeline bubbles because of hazards in MXP's front-end. Therefore, whenever a custom instruction has state (like computing the running sum of an entire vector, for instance), it must only update its state while the `vci_valid` signal is asserted for its particular function. The `vci_vector_start` signal indicates the start of a new vector when it is asserted and `vci_valid` is asserted; if `vci_valid` is 0 `vci_vector_start` should be ignored. If a 2D or 3D matrix instruction is used, `vci_vector_start` is not reasserted at the beginning of each row or matrix; it only signals the start of a new custom instruction. Similarly, `vci_vector_end` signal indicates the end a vector when `vci_valid` is also asserted. For pipelined VCIs (Pipelining Custom Instructions) no more valid data will be passed to the VCI after `vci_vector_end` is asserted; at this point all incoming data has been presented to the VCI and the last output data for the instruction is expected `VCUSTOMX_DEPTH` cycles later.

If the custom instruction does not need to modify the destination address (Modifying the Destination Address), `vci_dest_addr_in` and `vci_dest_addr_out` should be omitted from the VCI. For VCIs that do modify the detination address `vci_dest_addr_in` is the destination address that was generated by the address generation logic, while `vci_dest_addr_out` is the address that will actually be written to.

The *Instruction Modifiers* come directly from the instruction opcode and mode type and datasize specifier. An instruction of `vbx( VVHU, VCUSTOM1, dest, srcA, srcB );` would have `vci_signed=0`, and `vci_opsize=01`. If a datasize conversion is specified during a custom instruction, the data and `vci_opsize` signal are at the execution width; see the *MXP Programming Reference* for details.

The *Data Signals* are for flags, data, and their byte enables. Data is little endian, and vectors are always aligned to lane 0 of the custom instruction. `vci_byte_valid` has a bit for each input byte that indicates the byte contains valid data. Most custom instructions will simply pass the `vci_byte_valid` signal through a shift register (that has the same number of stages as the pipeline depth of the custom instruction) to the `vci_byteenable` output signal, or directly from `vci_byte_valid` to `vci_byteenable` if zero pipeline stages are used. `vci_byteenable` controls which bytes are written back, and might be modified to create custom conditional operations, for instance. `vci_data_a` and `vci_data_b` are the input data sources, and `vci_data_out` is the destination data. `vci_flag_a` and `vci_flag_b` are the input flags, and `vci_flag_out` is the destination flags. There is one flag per byte of data; when generating flags for halfword and word data types the desired flag value must be written to all flag bits associated with that halfword/word.

3 Example Custom Instructions

The following examples serve as a reference to how to implement functionalities with custom instructions.

3.1 Basic CI: A Implies B

This custom instruction implements A implies B, or in boolean logic (not A) or B. This allows the two instructions

```
vbx(SVXX, VXOR, dest, 0xFFFFFF, srcA); //dest = ~srcA
vbx(VVXX, VOR,  dest, dest,      srcB); //dest |= srcB
```

to be replaced with the single instruction

```
vbx(VVXX, VCUSTOM0, dest, srcA, srcB); //dest = (~srcA) | srcB
```

Note that since A implies B is a bitwise logical operation, the datasize and signedness instruction specifiers do not affect its operation, and so were left as `XX` in the example code.

A VHDL implementation of this custom instruction can be found named vci_a_implies_b. Within vci_a_implies_b.vhd there is a vci_a_implies_b entity. It has a single generic, `VCI_LANES`. It is good practice to always genericize your custom instruction to work with a configurable number of lanes even if it only uses a fixed number of lanes in the current system. `VCI_LANES` controls the width of several signals (see HDL Signals to/from Custom Instruction for a list of signals that depend on the number of lanes). If the width of these signals does not match the width of the signals to/from MXP (set by its own `VCI_LANES` parameter) synthesis of the system will fail.

The logic that computes A implies B takes the input data `vci_data_a` and `vci_data_b` and produces a result which is sent to the output.

```
    data_a_implies_b <= (not vci_data_a) or vci_data_b;

...

    vci_data_out      <= data_a_implies_b;
```

The `vci_flag_out` output is handled similarly. `vci_byteenable` is set to only write back data that was valid, and so it is set to the `vci_byte_valid` signal. This custom instruction does not need to modify the destination address, so the `vci_dest_addr_in` and `vci_dest_addr_out` signals are not included in the top level port list. Because it is not pipelined, the `VCUSTOMX_DEPTH` parameter corresponding to its opcode should be set to 0.

3.2 Deep Pipeline CI: Fixed-Point Square-Root

This custom instruction implements fixed-point square root. Details of the fixed-point square-root implementation can be found in the VHDL comments. The CI has a data pipeline `VCI_STAGES` stages deep. The pipeline stages for the data and flags are internal to the vendor provided square root core; for a Q16.16 configuration 25 stages are used in the Xilinx provided core and 16 in the Altera provided core. The byteenables are passed through but need to be delayed by the same number of stages, so a shift register `VCI_STAGES` deep is used.

```
process (vci_clk)
begin  -- process
  if vci_clk'event and vci_clk = '1' then  -- rising clock edge
    --Byte enable shifter; write back any valid inputs
    byteena_out_shifter(0) <=
      vci_byte_valid;
    byteena_out_shifter(byteena_out_shifter'left downto 1) <=
      byteena_out_shifter(byteena_out_shifter'left-1 downto 0);
  end if;
end process;
```

The `VCUSTOMX_DEPTH` parameter of MXP corresponding to the VCI's opcode should be set to `VCI_STAGES`.