ρ-VEX core manual

Jeroen van Straten

December 29, 2014

This document is just me spitting out documentation so far. Don't expect coherence just yet...

Contents

1 Introduction

The ρ-VEX core is an optionally runtime-reconfigurable VLIW processor based on the HP VEX architecture. Aside from being runtime-reconfigurable, the core has several design-time configurable parameters by means of generics, and many more are available through package constants. A coarse list of configuration parameters is listed below; this list is not exhaustive.

- Runtime-reconfigurable parameters[1]:
  - Number of lanes active.
  - Context to lane mapping: multiple contexts can be run at once, or a single context can run on all lanes.
- Design-time-configurable parameters through generics:
  - Number of lanes: 2, 4, 8 or 16.
  - Number of hardware contexts (to switch between and/or run in parallel): 1, 2, 4, 8 or 16.
  - Number of lane groups; determines the degree of reconfigurability.
  - Lane resource configuration: each lane within a group can optionally have a multiplier, and it is configurable which lane within the group has a memory interface and which can handle branch operations.
  - Long immediate forwarding options.
  - Whether forwarding logic is instantiated or not.
- Design-time-configurable parameters through package constants:
  - Instruction set and opcode mapping[2].
  - Pipeline and forwarding configuration, including memory bus latency: the timing of all functional units within the pipeline can be changed as much as the functional units permit, to augment the timing characteristics of the core without changing behavioral VHDL code.

todo: short history about the core; previous versions and references

This document serves as basic documentation for the ρ-VEX core files in `/lib/rvex/core`. More detailed (and probably more up-to-date) documentation can be found in the VHDL source code itself. This document does not provide documentation for the support packages within the `rvex` library.

[1]The degree in which these parameters are configurable depends on design-time configuration of the core.
[2]Modifications to the datapath require changes to behavioral VHDL code.

2 Overview

A block diagram of the rvex core is shown in Figure 1. The abbreviations used are described below. The same abbreviations are used in the code to mark the source and/or destination of a signal.

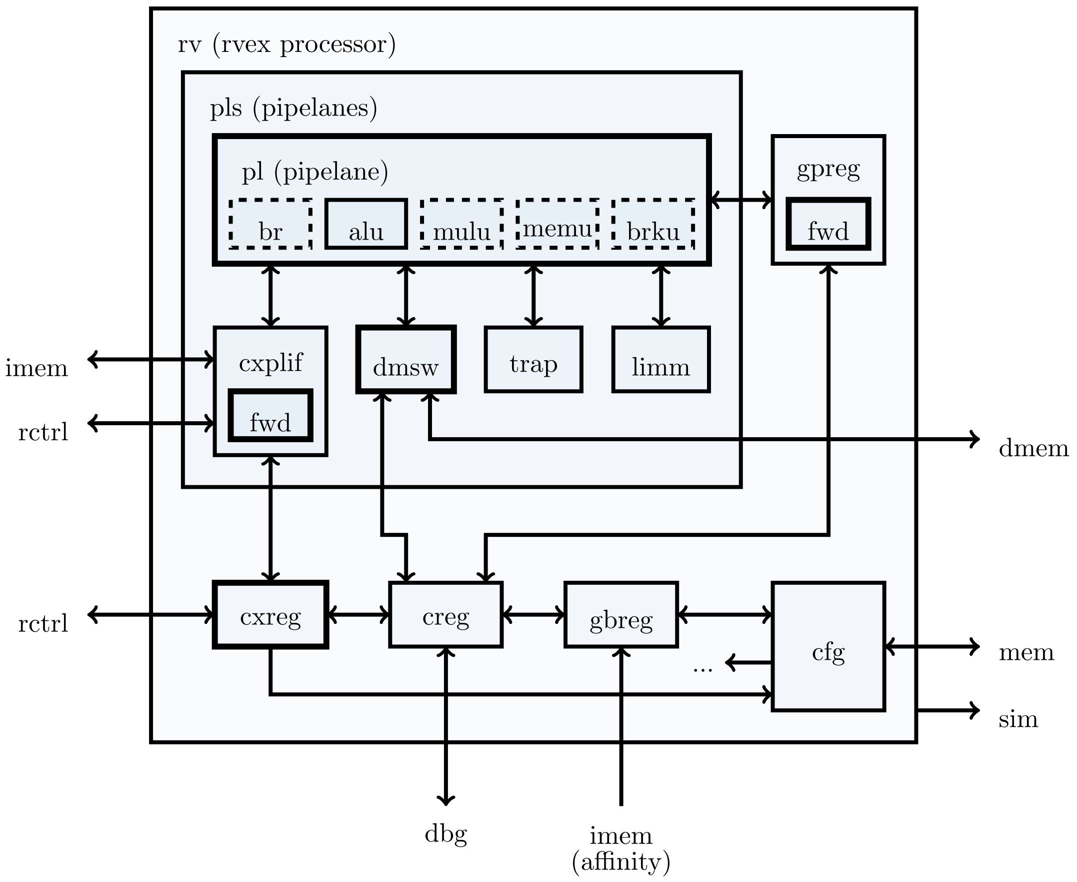

Figure 1: Block diagram of the ρ-VEX core.

ρ-VEX processor `rv` `core.vhd`

This is the toplevel file for the rvex processor.

External ρ-VEX package - `core_pkg.vhd`

This package contains type definitions and constants relevant both in the core internally and for the external interface of the core. In particular, it contains the type specification for the `CFG` generic, and the `rvex_cfg` subprogram which should be used to construct or modify it.

Internal ρ-VEX package - `core_intIface_pkg.vhd`

This package contains all type specifications, constants and subprograms which are relevant only to the core files and do not belong in a specific file. This file does not contain configuration constants, aside from the following constants related to simulation performance: `GEN_VHDL_SIM_INFO`, `SIM_FULL_GPREG_FILE` and `RVEX_UNDEF`. These constants are documented extensively in the code.

Pipelanes `pls` `core_pipelanes.vhd`

This entity contains the datapaths for the processor. It also contains the reconfigurable routing logic used to couple or decouple lane groups.

Pipelane `pl` `core_pipelane.vhd`

This entity contains the datapath for a single pipelane, capable of executing a single syllable. It instantiates the necessary functional units based on configuration and its index. The pipeline is described in a single behavioral process, in such a way that the timing can be modified by just changing constants. These constants may be found in `core_pipeline_pkg.vhd`. Assert statements are in place to check the configuration specified by `core_pipeline_pkg.vhd`.

Pipeline configuration - `core_pipeline_pkg.vhd`

This contains constants which define the timing and the number of stages of the pipeline. All stage definitions and latencies may be changed without breaking functionality, as long as the requirements for each constant are complied with. Assert statements in `core_pipelane.vhd` verify that the pipeline configuration is valid during static elaboration.

Branch unit `br` `core_branch.vhd`

This entity determines the program counter for the next cycle and provides its pipelane with the capability of executing branch class syllables. There must be exactly one branch unit in each pipelane group, and only the branch unit in the highest indexed pipelane group is active when groups are coupled.

Arithmetic logic unit `alu` `core_alu.vhd`

This entity contains the arithmetic unit for the ρ-VEX. The ALU takes up to two 32-bit integer operands and one boolean operand as input, and outputs a 32-bit integer and/or a boolean. Registers may be inserted in two places in the datapath, affecting the timing of the ALU. Insertion is controlled by `L_ALU1` and `L_ALU2`, defined in `core_pipeline_pkg.vhd`.

Memory unit `memu` `core_memu.vhd`

This entity interfaces with the data memory port and control registers of the ρ-VEX. When present, it provides its pipelane with the capability of executing memory class syllables. There must be exactly one memory unit in each pipelane group, and only the memory unit in the highest indexed pipelane group is active when groups are coupled.

Multiply unit `mulu` `core_mulu.vhd`

This entity contains a 32x16 multiplier. When present, it provides its pipelane with the capability of executing multiply class syllables. The latency is configurable using the `L_MUL` constant in `core_pipeline_pkg.vhd`. Directives for the Xilinx synthesis tools are in place to have the tools automatically insert the pipeline registers in the appropriate places within the DSP core.

Breakpoint unit `brku` `core_brku.vhd`

This entity matches the PC and memory access command against the up to four enabled hardware breakpoint registers, and generates debug traps if there is a match. It should only be instantiated in pipelanes which also have a memory unit.

Opcode package - `core_opcode_pkg.vhd`

This package defines the `OPCODE_TABLE` constant array, which maps the 8-bit opcode portion of a syllable to its control signals and (dis)assembly information. Operations can be added, removed and modified without breaking the processor. In addition, if the (dis)assembly formatting strings are maintained properly, the core unit test runner will generate correct machine code regardless of the opcode configuration when the `load` command is used, but externally compiled code loaded with the `srec` command will of course need to be recompiled.

Datapath ctrl. signal package
- `core_opcodeDatapath_pkg.vhd`

This package defines sets of control signals defining how the functional units are to be connected to the register files and each other, for use in `core_opcode_pkg.vhd`.

ALU ctrl. signal package - `core_opcodeAlu_pkg.vhd`

This package defines sets of control signals defining valid ALU functions for use in `core_opcode_pkg.vhd`.

Mult. ctrl. signal package - `core_opcodeMultiplier_pkg.vhd`

This package defines sets of control signals defining valid multiplier functions for use in `core_opcode_pkg.vhd`.

Mem. ctrl. signal package - `core_opcodeMemory_pkg.vhd`

This package defines sets of control signals defining valid memory unit functions for use in `core_opcode_pkg.vhd`.

Branch ctrl. signal package - `core_opcodeBranch_pkg.vhd`

This package defines sets of control signals defining valid branch unit functions for use in `core_opcode_pkg.vhd`.

(Dis)assembler package - `core_asDisas_pkg.vhd`

This package defines simulation/elaboration only subprograms which can perform basic assembly and disassembly, used for the core unit test runner and the simulation-only core state output signal.

General purpose registers `gpreg` `core_gpRegs.vhd`

This entity instantiates the general purpose register file and associated forwarding logic. Two register file implementations are available, specified in `core_gpRegs_sim.vhd` and `core_gpRegs_mem.vhd`.

BRAM-based `gpreg` **spec.** - `core_gpRegs_mem.vhd`

This entity specifies the general purpose register file in such a way that block RAMs are inferred for the register contents. In order to provide simultaneous access to all read and write ports, the contents of the register file are duplicated for each read and write port pair. Fabric based registers are used to store which write port last wrote to each register. Because of the duplication, it is hard to trace the contents of the register file in simulation. Therefore, `core_gpRegs_sim.vhd` will be used for simulation instead, unless `SIM_FULL_GPREG_FILE` in `core_intIface_pkg.vhd` is set to `true`.

Behavioral `gpreg` **spec.** - `core_gpRegs_sim.vhd`

This entity specifies the general purpose register file behaviorally, in such a way that the register contents can be easily traced in simulation. However, it is not decently synthesizable.

Forwarding logic `fwd` `core_forward.vhd`

This entity infers the priority encoder and muxes needed for forwarding. It is highly generic and customizable, allowing it to be used for both the general purpose register file as well as the branch register file and link register.

Context-pipelane interface `cxplif` `core_contextPipelaneIFace.vhd`

This entity serves as a central reconfigurable routing matrix between all pipelanes, pipelane groups and contexts.

Data memory switch `dmsw` `core_dmemSwitch.vhd`

This entity switches between forwarding data memory access requests to the external data memory port and the core control registers. It adds additional latency stages to the core control register read data signal when the memory latency (`L_MEM` in `core_pipeline_pkg.vhd`) is specified to be greater than one.

Long immediate routing `limm` `core_limmRouting.vhd`

This entity contains the inter-pipelane routing needed to support long immediate (`LIMMH`) instructions. It can be configured to support routing long immediate data between aligned pipelane pairs and/or from the previous pair by means of the `limmhFromNeighbor` and `limmhFromPreviousPair` members of the `CFG` generic. When `limmhFromNeighbor` is set, a `LIMMH` instruction in lane 0 can forward to lane 1, lane 1 can forward to lane 0, lane 2 to 3, 3 to 2 and so on. When `limmhFromPreviousPair` is set, lane 0 can forward to lane 2, lane 1 to 3, 3 to 4 and so on. Registers are instantiated in this case when the generic binary bundle size (`genBundleSizeLog2` in `CFG`) is set to be larger than the size of a lane group to store the `LIMMH` data from the previous instruction when necessary.

Trap routing `trap` `core_trapRouting.vhd`

This entity merges trap information from pipelanes together when they are coupled, and broadcasts the merged information back to the lanes for the invalidation logic.

Trap code package - `core_trap_pkg.vhd`

This package defines the `TRAP_TABLE` constant array, which maps an 8-bit trap cause code to its friendly name and control signals. Constants are also available for each trap cause, prefixed by `RVEX_TRAP_`. In general, the definitions in this package can be changed without breaking functionality, as long as existing traps are not removed.

Control registers `creg` `core_ctrlRegs.vhd`

This entity contains the generic parts of the core control registers and the bus logic needed to access them. Functionality of the registers is defined by `core_contextRegLogic.vhd` and `core_globalRegLogic.vhd`, and the word addresses of the registers are defined in `core_ctrlRegs_pkg.vhd`.

Ctrl. reg. map package - `core_ctrlRegs_pkg.vhd`

This package defines the constants which determine the memory map of the core control registers and several boilerplate subprograms used to generate certain types of control registers with. All control register memory map related constants prefixed with `CR_` can be changed without breaking code, as long as the overal map remains consistent. Note that the constant specifying at which word the global registers stop and the context-specific registers start, `CTRL_REG_GLOB_WORDS`, is defined in `core_intIface_pkg.vhd`.

Ctrl. reg. bank - `core_ctrlRegs_bank.vhd`

This entity instantiates a group of generic control registers. Used by `core_ctrlRegs.vhd` to instantiate the global control registers and the context control registers for each context.

Ctrl. reg. read port - `core_ctrlRegs_readPort.vhd`

This entity instantiates an additional read port for a control register bank instantiated with `core_ctrlRegs_bank.vhd`.

Ctrl. reg. bus switch - `core_ctrlRegs_busSwitch.vhd`

This entity connects a single control register bus master to several slaves, switching based on the request address.

Ctrl. reg. context switch - `core_ctrlRegs_contextLaneSwitch.vhd`

This entity connects the control register busses from the memory units to the appropriate context-specific register bank, based on the current runtime configuration.

Context register logic `cxreg` `core_contextRegLogic.vhd`

This entity defines the functionality of the context-specific control registers. Extensive documentation is provided in the code, and the registers are defined using the subprograms defined in `core_ctrlRegs_pkg.vhd`. This should make it easy to define new control registers and understand the current ones.

Global register logic `gbreg` `core_globalRegLogic.vhd`

Similar to `core_contextRegLogic.vhd`, this entity defines the functionality of the global core control registers. Be aware that these registers are only writable from the external debug bus interface.

Configuration control `cfg` `core_cfgCtrl.vhd`

This entity provides the runtime configuration control signals to the rest of the core and controls the timing for runtime reconfiguration.

Configuration decoder `-` `core_cfgCtrl_decode.vhd`

This entity error-checks and decodes the configuration words as presented by the core or the debug bus when they request reconfiguration.

Memory `mem` `-`

Abbreviation for the memory or cache connected to the ρ-VEX.

Instruction memory `imem` `-`

Abbreviation for the instruction memory or cache connected to the ρ-VEX.

Data memory `dmem` `-`

Abbreviation for the data memory or cache connected to the ρ-VEX.

Debug bus `dbg` `-`

Abbreviation for the debug bus connected to the ρ-VEX.

Run control `rctrl` `-`

Abbreviation for the entity or group of entities connected to the ρ-VEX run control interface.

Simulation `sim` `-`

Abbreviation for the behavioral VHDL simulator, used as destination for simulation-only debug output signals.

3 Naming conventions

All entities, packages, labels, types and hierarchy-crossing signals use the following naming conventions.

- All signals, entity names, package names and types use a combination of camelCase and underscores. Typically, underscores are used as a form of hierarchy separation, where the VHDL language does not otherwise allow it, and camelCase is used to indicate word boundaries within one level of hierarchy. For example, `core_ctrlRegs_bank` refers to a (register) bank for the control registers for the ρ-VEX core.
- Most signal names start with an underscore-terminated abbreviation which indicates the source and destination entity. This identifier contains two entity abbreviation codes separated by a `2`. The entity abbreviation codes are defined at the top of `core.vhd` and are also listed in Section 2. Sometimes other abbreviations are used for signals local to one entity, which should be clear from context.
- All constant names are uppercase with underscores.
- Labels typically use underscores only, to prevent conflicts between similarly named entities and signals.
- Types use one of the following suffixes to indicate what kind of type they represent.
  - `_type`: scalar type.
  - `_array`: array type.
  - `_ptr`: access type for a scalar.
  - `_array_ptr`: access type for an array.
- Enties and packages have the same name as their filename; exactly one entity or package is defined per VHDL file. Package names end in `_pkg`. All ρ-VEX core files start with `core_` to keep their names unique within the `rvex` package, which contains a number of supporting packages as well.

4 Instantiation and interface

The following listing serves as an instantiation template for the core. The code is documented in the following sections.

```vhdl
library rvex;
use rvex.common_pkg.all;
use rvex.core_pkg.all;

-- ...

rvex_inst: entity rvex.core
  generic map (

    -- Core configuration.
    CFG => rvex_cfg(
      numLanesLog2              => 3,
      numLaneGroupsLog2         => 2,
      numContextsLog2           => 2
      -- ...
    )

  )
  port map (

    -- System control.
    reset                       => reset,
    clk                         => clk,
    clkEn                       => clkEn,

    -- Run control interface.
    rctrl2rv_irq                => rctrl2rv_irq,
    rctrl2rv_irqID              => rctrl2rv_irqID,
    rv2rctrl_irqAck             => rv2rctrl_irqAck,
    rctrl2rv_run                => rctrl2rv_run,
    rv2rctrl_idle               => rv2rctrl_idle,
    rctrl2rv_reset              => rctrl2rv_reset,
    rv2rctrl_done               => rv2rctrl_done,

    -- Common memory interface.
    rv2mem_decouple             => rv2mem_decouple,
    mem2rv_blockReconfig        => mem2rv_blockReconfig,
    mem2rv_stallIn              => mem2rv_stallIn,
    rv2mem_stallOut             => rv2mem_stallOut,

    -- Instruction memory interface.
    rv2imem_PCs                 => rv2imem_PCs,
    rv2imem_fetch               => rv2imem_fetch,
    rv2imem_cancel              => rv2imem_cancel,
    imem2rv_instr               => imem2rv_instr,
    imem2rv_affinity            => imem2rv_affinity,
    imem2rv_fault               => imem2rv_fault,

    -- Data memory interface.
    rv2dmem_addr                => rv2dmem_addr,
    rv2dmem_readEnable          => rv2dmem_readEnable,
    rv2dmem_writeData           => rv2dmem_writeData,
    rv2dmem_writeMask           => rv2dmem_writeMask,
    rv2dmem_writeEnable         => rv2dmem_writeEnable,
    dmem2rv_readData            => dmem2rv_readData,
    dmem2rv_fault               => dmem2rv_fault,

    -- Control/debug bus interface.
    dbg2rv_addr                 => dbg2rv_addr,
    dbg2rv_readEnable           => dbg2rv_readEnable,
    dbg2rv_writeEnable          => dbg2rv_writeEnable,
    dbg2rv_writeMask            => dbg2rv_writeMask,
    dbg2rv_writeData            => dbg2rv_writeData,
    rv2dbg_readData             => rv2dbg_readData

  );
```

4.1 Configuration vector

The `CFG` generic is used to configure the ρ-VEX core. A function from the `core_pkg` package is used to generate this record in such a way that code instantiating the core will be forward compatible: because the parameters in the function are optional, parameters can be added without breaking old instantiation code.

The instantiation template does not list all the configuration parameters available. For an up-to-date list, refer to the comments in the `rvex_generic_config_type` type declaration in `core_pkg`.

4.2 Signal types

The following custom VHDL types are used in the interface signals. They are defined in `common_pkg`.

```vhdl
subtype rvex_address_type      is std_logic_vector(31 downto  0);
subtype rvex_data_type         is std_logic_vector(31 downto  0);
subtype rvex_mask_type         is std_logic_vector( 3 downto  0);
subtype rvex_syllable_type     is std_logic_vector(31 downto  0);

type rvex_address_array        is array (natural range <>) of rvex_address_type;
type rvex_data_array           is array (natural range <>) of rvex_data_type;
type rvex_mask_array           is array (natural range <>) of rvex_mask_type;
type rvex_syllable_array       is array (natural range <>) of rvex_syllable_type;
```

The `address`, `data` and `mask` types are used for the data bus interfaces, wherein `mask` is used for a per-byte write mask. The `syllable` types represent a syllable, i.e., an instruction word for a single lane.

4.3 System control

The system control signals include the clock source for the core, a synchronous reset signal and a global clock enable signal. `clk` and `reset` are required `std_logic` input signals. `clkEn` is an optional `std_logic` input signal.

The core is clocked on the rising edge of `clk` while `clkEn` is high. When a rising edge on `clk` occurs while `reset` is high, all resettable components of the core will be reset, regardless of the state of `clkEn`. Only block and distributed RAM resources are not reset, because this is not possible on Xilinx FPGAs without reconfiguration. In particular, this means that the general purpose registers are not reset to 0 by asserting `reset`, thus their contents should be considered to be undefined when a program starts. `r0.0` is an exception to this rule, as it is always zero.

4.4 Run control

The run control signals provide an interface between the core and an interrupt controller and/or host controller. All signals are optional. All signals are arrays of some sort, indexed by hardware context IDs in descending order.

- `rctrl2rv_irq : in std_logic_vector(`*number of contexts - 1* `downto 0)`
- `rctrl2rv_irqID : in rvex_address_array(`*number of contexts - 1* `downto 0)`
- `rv2rctrl_irqAck : out std_logic_vector(`*number of contexts - 1* `downto 0)`

  When `rctrl2rv_irq` is high, an interrupt trap will be generated within the indexed context as soon as possible, if the interrupt enable flag in the context control register is set. Interrupt entry is acknowledged by `rv2rctrl_irqAck` being asserted high for one `clkEnabled` cycle. `rctrl2rv_irqID` is sampled in exactly that cycle and is made available to the trap handler through the trap argument register. When not specified, `rctrl2rv_irq` is tied to `'0'` and `rctrl2rv_irqID` is tied to `X"00000000"`.

When `rv2rctrl_irqAck` is high, an interrupt controller would typically release `rctrl2rv_irq` and set `rctrl2rv_irqID` to a value signalling that no interrupt is active. Alternatively, if more interrupts are pending, `rctrl2rv_irq` may remain high and `rctrl2rv_irqID` may be set to the code identifying the next interrupt.

Releasing `rctrl2rv_irq` while `rv2rctrl_irqAck` is *not* high may lead to unexpected behavior. Because it takes time for traps to propagate through the pipeline, interrupt traps are issued before they are acknowledged, after which they cannot be stopped anymore. Therefore, the core might acknowledge an interrupt which was recently cleared while interrupts are enabled, and enter the trap handler accordingly.

Acknowledging the interrupt and registering the ID when the trap is issued is not a viable option, because the trap might theoretically be overruled by a regular trap issued by a previous instruction in a later pipeline stage, in which case it would be possible for interrupts to be lost. The stage in which interrupt traps are issued is defined by the stage in which the application can update the interrupt enable flag.

This behavior may be handled by defining a certain interrupt ID as a cleared interrupt, which is serviced by the trap handler as no operation. In addition, the behavior may be prevented altogether by only clearing interrupts while interrupts are disabled.

- `rctrl2rv_run : in std_logic_vector(`*number of contexts - 1* `downto 0)`
- `rv2rctrl_idle : out std_logic_vector(`*number of contexts - 1* `downto 0)`

  When `rctrl2rv_run` is asserted low, the indexed context will stop executing instructions as soon as possible. It will finish instructions which were already in the pipeline and have already committed data, and set the program counter to point to the next instruction which should be issued for the program to be resumed correctly. When the context has finished pausing, `rv2rctrl_idle` will be asserted high. This may also happen while `rctrl2rv_run` is low, when the core is being halted for a different reason (for example, to prepare for reconfiguration, because a context is not configured to run, etc.).

  When `rctrl2rv_run` is not specified, it is tied to `'1'`.

  Note that per this definition of idle, a context is not necessarily idle when it is stalled, because instructions may still be pending completion in the pipeline.

- `rctrl2rv_reset : in std_logic_vector(`*number of contexts - 1* `downto 0)`
- `rv2rctrl_done : out std_logic_vector(`*number of contexts - 1* `downto 0)`

  When `rctrl2rv_reset` is asserted, the context control registers for the indexed context are synchronously reset in the next `clkEn`abled cycle. When it is not specified, it is tied to `'0'`. `rv2rctrl_done` will be asserted high when a `stop` instruction is encountered.

  When the ρ-VEX is running as a co-processor, `rctrl2rv_reset` could be used as an active low flag indicating that the currently loaded kernel needs to be executed, in which case `rv2rctrl_done` signals completion.

4.5 Common memory interface

These control signals are common to both the data and instruction memory interface.

- `rv2mem_decouple : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  This vector represents the current runtime configuration of the core. In particular, it specifies which lane groups are working together to execute code within a single context. When a bit in this vector is high, the indexed lane group is "decoupled" from the next lane group, i.e., is operating within a different context. When a bit is low, the indexed lane group is working as a slave to the next higher indexed lane group for which the bit is set.

  Due to constraints in the core, the indices of pipelane groups working together are always aligned to the number of pipelane groups in the group. As an example, if pipelane groups 0 and 1 are working together, group 2 cannot join them without group 3 also joining them. This allows binary tree structures to be used in the coupling logic. This means that, in the default core configuration, only the following decouple vectors are legal: `"1111"`, `"1110"`, `"1011"`, `"1010"` and `"1000"`.

  The state of the `rv2mem_decouple` signal has several implications on the behavior of the memory ports on the ρ-VEX.

  - The data memory ports associated with lane groups for which the `rv2mem_decouple` signal is low will not make requests, and may thus be ignored if this is favourable for the memory implementation.
  - The PCs presented by the instruction memory ports will always be contiguous and aligned for groups which are working together. The fetch and cancel signals will always be identical.

  In addition, the ρ-VEX assumes that the `mem2rv_blockReconfig` and `mem2rv_stallIn` signals are identical for coupled pipelane groups. Undefined behavior will result if these assumptions are violated.

- `mem2rv_blockReconfig : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  This signal can be used by the memories to block reconfiguration due to ongoing operations. When a bit in this vector is high, the indexed group will not split up or merge with other groups. The ρ-VEX will assume that the associated bits in the `mem2rv_blockReconfig` signal will be released eventually when no operations are requested by those pipelane groups. When pipelane groups are coupled, their respective `mem2rv_blockReconfig` signals must be identical. When this signal is not specified, it is tied to all zeros.

- `mem2rv_stallIn : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Stall input signals for each pipelane group. When the stall signal for a pipelane group is high, it will behave as if it is clock gated. When pipelane groups are coupled, their respective `mem2rv_stallIn` signals must be identical. When this signal is not specified, it is tied to all zeros.

- `rv2mem_stallOut : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Stall output signals for each pipelane group. This serves as a combined stall signal from all stall sources, indicating whether a pipelane group is actually stalled or not. When `rv2mem_stallOut` is high, all memory request signals from the associated pipelane group should be considered to be undefined. Memory access requests should thus be initiated (and registered) only at the rising edge of the `clk` signal when `clkEn` is high and the associated `rv2mem_stallOut` signal is low.

When pipelane groups are coupled, their respective `rv2mem_stallOut` signals will be equal. In addition, the `unifiedStall` configuration parameter in the `CFG` record may be set to `true` to enforce equal stall signals for all pipelane groups at all times, should this be desirable for the memory implementation.

4.6 Instruction memory interface

These signals interface between the ρ-VEX and the instruction memory or cache. All signals in this section are clock gated by not only `clkEn`, but also by the respective signal in `rv2mem_stallOut`. They should be considered to be invalid when the respective `rv2mem_stallOut` signal is high. The number of enabled clock cycles after which the reply for a request is assumed to be valid is defined by `L_IF`, which is defined in `core_pipeline_pkg`. `L_IF` defaults to 1.

- `rv2imem_PCs : out rvex_address_array(`*number of lane groups - 1* `downto 0)`

  Program counter outputs for each lane group. These will always be aligned to the size of an instruction for a full lane group. When lane groups are coupled, the PC for the first lane group will always be aligned to the size of the instruction to be executed on the set of lane groups, and the PCs for those lang groups will be contiguous.

- `rv2imem_fetch : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Read enable output. When high, the instruction memory should supply the instructions pointed to by `rv2imem_PCs` on `imem2rv_instr` after `L_IF` processor cycles. `L_IF` is set in `core_pipeline_pkg.vhd` and defaults to 1.

- `rv2imem_cancel : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Cancel signal. This signal will go high combinatorially (regardless of the stall input from the memory) when it has been determined that the instruction requested for this cycle will not be used. In this case, the memory may cancel the request in order to be able to release the stall signal earlier. This signal can safely be ignored for correct operation.

- `imem2rv_instr : in rvex_syllable_array(`*number of* lanes - *1* `downto 0)`

  Syllable input for each lane. Expected to be valid `L_IF` processor cycles after `rv2imem_fetch` is asserted if `rv2imem_cancel` and `imem2rv_fault` are low.

- `imem2rv_affinity : in std_logic_vector(`$n \log(n)$ - *1* `downto 0)`
  Where n = number of lane groups

  Optional block affinity input signal for reconfigurable caches. If used, it is expected to have the same timing as the `imem2rv_instr` signal. Each lane group has log(*number of lane groups*) bits in this signal, forming an unsigned integer which indexes the lane group which serviced the instruction read. When the processor wants to reconfigure, it may use this signal as a hint to determine which program should be placed on which lane group next, assuming that there will be fewer cache misses if the currently running application is mapped to the lane group indexed by the affinity signal.

- `imem2rv_fault : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Instruction fetch fault input signal. Expected to have the same timing as the `imem2rv_instr` signal. When high, an instruction fetch trap is generated and the instruction defined by `imem2rv_instr` will not be executed.

4.7 Data memory interface

These signals interface between the ρ-VEX and the data memory or cache. All signals in this section are clock gated by not only `clkEn`, but also by the respective signal in `rv2mem_stallOut`. They should be considered to be invalid when the respective `rv2mem_stallOut` signal is high. The number of enabled clock cycles after which the reply for a request is assumed to be valid is defined by `L_MEM`, which is defined in `core_pipeline_pkg`. `L_MEM` defaults to 1.

- `rv2dmem_addr : out rvex_address_array(`*number of lane groups - 1* `downto 0)`

  Memory address which is to be accessed if `rv2dmem_readEnable` or `rv2dmem_writeEnable` is high. The two least significant bits of the address will always be `"00"` and may be ignored. Note that a configurable 128 byte block within this 4 GiB memory space is inaccessible, because it is replaced by the core control registers. This is configurable through the `cregStartAddress` entry in `CFG`, which defaults to 0xFFFFFF80, meaning that 0xFFFFFF80..0xFFFFFFFF is inaccessible.

- `rv2dmem_readEnable : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Active high read enable signal from the core for each memory unit. This signal will never be asserted when the respective signal in `rv2mem_decouple` is low. When high during an enabled rising clock edge, the ρ-VEX expects the access result to be valid `L_MEM` enabled cycles later.

- `rv2dmem_writeData : out rvex_data_array(`*number of lane groups - 1* `downto 0)`
- `rv2dmem_writeMask : out rvex_mask_array(`*number of lane groups - 1* `downto 0)`

  These signals define the write operation to be performed when `rv2dmem_writeEnable` is high. `rv2dmem_writeMask` contains a bit for each byte in `rv2dmem_writeData`, which determines whether the byte should be written or not: when high, the respective byte should be written; when low, the byte should not be affected. Mask bit i governs data bits $i * 8 + 7$ downto $i * 8$.

- `rv2dmem_writeEnable : out std_logic_vector(`*number of lane groups - 1* `downto 0)`

  Active high read enable signal from the core for each memory unit. This signal will never be asserted when the respective signal in `rv2mem_decouple` is low. When high during an enabled rising clock edge, the ρ-VEX expects either that the write request defined by `rv2dmem_addr`, `rv2dmem_writeData` and `rv2dmem_writeMask` will be performed, or that `dmem2rv_fault` is asserted high `L_MEM` cycles later.

- `dmem2rv_readData : in rvex_data_array(`*number of lane groups - 1* `downto 0)`

  This is expected to contain the read data for read requested by `rv2dmem_readEnable` and `rv2dmem_addr` `L_MEM` enabled cycles earlier, unless `dmem2rv_fault` is high, indicating that a bus fault occurred.

- `dmem2rv_fault : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  This is expected to be valid `L_MEM` enabled cycles after a read or write request. When high, this indicates that the read or write could not be performed for some reason. In this case, a `DMEM_FAULT` trap will be issued. The trap argument will be set to the address which was requested.

4.8 Control/debug bus interface

The debug bus provides an optional slave bus interface capable of accessing most of the registers within the core.

- `dbg2rv_addr : in rvex_address_type`
- `dbg2rv_readEnable : in std_logic`
- `dbg2rv_writeEnable : in std_logic`
- `dbg2rv_writeMask : in rvex_mask_type`
- `dbg2rv_writeData : in rvex_data_type`
- `rv2dbg_readData : out rvex_data_type`

  Debug interface bus. `dbg2rv_readEnable` and `dbg2rv_writeEnable` are active high and should not be active at the same time. `rv2dbg_readData` is valid one cycle after `dbg2rv_readEnable` is asserted and contains the data read from `dbg2rv_addr` as it was while `dbg2rv_readEnable` was asserted. `dbg2rv_writeMask`, `dbg2rv_writeData` and `dbg2rv_addr` define the write request when `dbg2rv_writeEnable` is asserted. All input signals are tied to `'0'` when not specified.