ρ-VEX user manual

Jeroen van Straten, TU Delft

September 25, 2017

Version: `a7549bc`
Core version tag: `ujA-L0P`

Contents

4 Control registers

1 Introduction

This manual intends to document the ρ-VEX reconfigurable VLIW processor. It is intended for software developers using the processor and hardware developers who are only interested in instantiating the processor in their system. It does not aim to document the internal workings of the processor; the comments in the source code are a better source of documentation for this. It also does not document the design choices that were made in the construction of the core; for this, readers are referred to [1].

The next chapter gives a top-level overview of the processor. The third chapter documents the instruction set architecture (ISA) in detail. The fourth chapter lists all the control registers of the processor. The fifth chapter handles the trap and interrupt system of the processor. The sixth handles reconfiguration, the component of the ρ-VEX that makes it special. The seventh documents how the core may be debugged using a computer. The final two chapters are intended for the hardware developers only, documenting the design-time configuration options of the core and how it may be instantiated.

Overview of the ρ-VEX processor

2

Let us begin by defining some terminology. The ρ-VEX processor is a Very Large Instruction Word (VLIW) processor, which means that each instruction can specify multiple independent operations. Such operations are called *syllables*; a full instruction is called a *bundle*. *Instruction* may be used for either a bundle or a syllable, depending on context. A VLIW processor capable of executing n syllables per cycle is called an *n-way* VLIW processor.

Because the amount of syllables in a bundle is usually[1] not fixed, the processor needs a way to tell which syllables belong to which bundle. In the VEX architecture, this is done by means of a *stop bit* in each syllable. If the stop bit is set, the next syllable in the program starts a new bundle. Otherwise, the next syllable is part of the same bundle.

When a VLIW processor executes a bundle, each syllable will be routed to its own *(pipe)lane*. Note the 'a' in lane; this is not a typo for pipeline (although each pipelane, confusingly, does contain its own pipeline). In other words, the pipelane is the thing that contains the computational resources to execute a syllable.

2.1 Reconfiguration

What makes the ρ-VEX processor special compared to other VLIW processors, is that while the total number of pipelanes is obviously fixed, the pipelanes can be distributed between different programs, running in parallel. This distribution can be changed at runtime by means of *reconfiguration*.

Note that 'reconfiguration' here is used to describe a process within the system described by a single FPGA bitstream. In other words, the FPGA bitstream does not need to be fully or partially reloaded when the ρ-VEX processor reconfigures itself. This allows reconfiguration to be done in a single cycle in theory, although it comes at the cost of needing FPGA slice muxes or LUTs to permit reconfiguration, instead of using the FPGA fabric directly.

Not all pipelanes are seperable by means of reconfiguration. Groups of inseperable pipelanes are called *lane groups*. Sometimes they are also referred to as *lanepairs* when a lane group contains two pipelanes, which is the most common configuration.

In order to be able to run multiple programs on a single ρ-VEX processor core at the same time, an ρ-VEX processor supports multiple *contexts*. Formally, a context contains the complete state of a program, from program counter to register file. However, a

[1] It is uncommon for the compiler to find enough parallelism in a program to fill an entire bundle. Therefore, if the bundle size is fixed, a lot of syllables will be NOP. While a fixed bundle size results in much simpler hardware, the size of the binary will be excessive. While main memory footprint is not so much an issue nowadays, memory throughput and latency is; the efficiency of the instruction coding directly affects execution speed as the memory is usually the bottleneck.

more useful way to think of ρ-VEX contexts is as virtual processor cores. By means of reconfiguration, the amount of lane groups dedicated to each virtual core can be changed. In fact, it is possible to completely pause such a virtual core by simply assigning zero lane groups to it.

2.2 Generic binaries

To compile for a VLIW processor, the compiler needs to be aware of what the maximum number of syllables per bundle is. However, reconfiguration changes this value at runtime, which would imply that each program should be compiled multiple times, for each bundle size possible with reconfiguration. This would severely limit the usefulness of reconfiguration, as it would be extremely difficult to reconfigure in the middle of program execution. At best, the program counters would be the only things that would not match between the two binaries.

The solution to this problem is a *generic binary* [2]. Generic binaries are compiled for the largest possible bundle size at which they may execute, referred to as the *generic bundle size*. This allows the compiler to extract as much parallelism as may ever be used. The difference between a normal binary compiled for the generic bundle size and a generic binary lies in additional rules imposed to the program by the assembler. These rules are carefully picked to ensure that, for instance, a bundle with four syllables in it still runs correctly if the two syllable pairs are run sequentially. Unless otherwise specified, an ρ-VEX generic binary refers to a binary compiled such that it runs correctly on 8-way, 4-way and 2-way ρ-VEX processor cores.

2.3 Intended applications

On the short term, the current version of the ρ-VEX processor is still primarily intended for research. The VHDL is written in a highly flexible and configurable way, thus making modifications for experiments relatively easy. At the same time, several complex features have been added to the core, in order to make it possible to, for instance, run Linux on it. Most notably, precise trap support has been added since the previous ρ-VEX version, necessary for adding a memory-management unit.

This combination of flexibility and complexity comes at a cost: speed. The current version of the ρ-VEX processor only runs at 37.5 MHz on a high-end Virtex 6 FPGA using the default configuration, while almost completely filling it up. Much more interesting is what the ρ-VEX architecture is capable of on the long run when better optimized, or even ported to an ASIC.

In general, VLIW processors are well-suited for executing highly parallel programs, such as those found in digital signal processing (DSP). In particular, the reconfiguration capabilities of the ρ-VEX processor allow it to be used in places where multiple DSP algorithms run in parallel in a real-time system, such that each task has its own deadlines.

To demonstrate, consider a hypothetical audio/video decoder DSP with the following characteristics as an example.

- The audio and video decoders do not depend on each other and can thus be executed in parallel. The decoders themselves are not multithreaded.
- Both tasks run 1.5x as fast when running on a 4-way VLIW compared to a 2-way VLIW.
- The execution times of both tasks are data dependent. For example, if there is a lot of movement in the video, then the video task will take longer to complete.
- It is possible to heuristically predict whether or not either decoder will meet its deadline at its current execution speed before the deadline, in a way that does not cost an excessive amount of additional computation. This can be done, for example, by decoding audio and video a few frames in advance, and assuming that if the current frame is computationally intensive, the next one will probably be too (locality).
- The audio task takes priority over the video task, as choppy audio is perceived as more intrusive than choppy video.
- For simplicity, assume that while the video decoder is decoding a single frame, the audio decoder has to decode a frame's worth of audio samples. In other words, the audio and video decoding tasks start at the same time and have the same deadline. In addition, assume that both tasks need an approximately equal amount of processing time for a single frame.

Let us now analyze the performance of this system if it were implemented on two 2-way VLIW processors. Each processor is simply assigned to one of the tasks. The primary downside to this system in the context of this discussion is that if the audio is overly complex, the audio decoder will miss its deadline, regardless of the whether the video processor was fully utilized or not.

To prevent this from happening, one may instead choose to implement the system on a single 4-way VLIW with a real-time operating system (RTOS) kernel. Notice that this system has the same amount of compute resources as the previous system. Now, the RTOS will ensure that the audio decoder runs before the video decoder. Because the audio decoder runs 1.5x as fast, it will likely meet its deadline now. While unlikely, it is possible that the video decoder will also complete in time now, but even if it does not, choppy video was considered favorable over choppy audio. The major downside of this system is that it is effectively much slower than the 2x2-way system, as the decoders do not actually run twice as fast when given twice as many computational resources, as the instruction level parallelism just is not always there.

The power of the ρ-VEX processor is that it can basically switch between these two implementations at runtime, depending on the actual load each task experiences. When neither task is in danger of failing to meet its deadline, the ρ-VEX processor could run in 2x2-way mode. However, if one of the tasks starts falling behind the other because it is more computationally intensive, the ρ-VEX processor could reconfigure to 1x4-way mode for that task. When it catches up, it will switch back to 2x2-way mode, as that is more efficient.

3 Instruction set architecture

The instruction set architecture (ISA) of the ρ-VEX processor is primarily based on VEX, an example VLIW architecture used in [3] to explain VLIW concepts. A compiler was developed for this architecture by HP, which is available as a free download for noncommercial use [4]. In addition, some enhancements were made to the instruction set to be compatible with the Lx architecture, introduced in [5]. Lx was designed by HP and STMicroelectronics for use in SoC (system-on-chip) designs, among which is the ST200 family of processors. This architecture comes with an Open64-based compiler and Linux port, available for download under the GNU GPL [6]. These tools together provide the basis for the toolchain used for the ρ-VEX. The instruction encoding for this version of the ρ-VEX processor is based on that of the previous major version of the ρ-VEX [7].

Instead of noting the differences between these architectures, this section functions as a reference for the ρ-VEX processor ISA in its current state, to save the reader from cross-referencing.

3.1 Assembly syntax

The following listing shows the syntax for a single instruction bundle.

```
start:
  c0 stw  0x10[$r0.1] = $r0.53
  c0 add  $r0.3       = $r0.0, -32
  c0 and  $b0.2       = $r0.0, $r0.10
  c0 call $l0.0       = interrupt
;;
```

The first line represents a label, as it ends in a colon. Each non-empty line that does not start with a semicolon and is not a label represents a syllable. The first part of the syllable, `c0`, is optional. It specifies the cluster that the syllable belongs to. Since the ρ-VEX processor currently does not support clusters, only cluster zero is allowed if specified. The second part represents the opcode of the syllable, defining the operation to be performed. The third part is the parameter list. Anything that is written to is placed before the equals sign, anything that is read is placed after. Finally, a double semicolon is used to mark bundle boundaries.

The syntax for a general purpose register is `$r0.`*index*, where *index* is a number from 0 to 63. The first 0 is used to specify the cluster, which, again, is not used in the ρ-VEX processor. Branch registers and the link register have the same syntax, substituting the 'r' with a 'b' or an 'l' respectively. The *index* for branch registers ranges from 0 to 7. For link registers only 0 is allowed.

Most instructions also accept a literal as their second operand. Literals may be a decimal or hexadecimal number (using `0x` notation), a label reference, or a basic C-

like integer expression. Literals represent 32-bit values with undefined signedness, i.e., `0xFFFFFFFF` and -1 specify the same value.

Finally, the load and store instructions require a memory reference as one of their operands. Memory references use the following syntax: *literal*`[$r0.`*index*`]`. At runtime, the literal is added to the register value to get the address, i.e. base + offset addressing is used.

A port of the GNU assembler (`gas`) is used for assembly. Please refer to its manual for information on target-independent directives or more information on the expressions mentioned above.

In general, the C preprocessor is used to preprocess assembly files. This allows usage of the usual C-style comments, includes, definitions, etc. In particular, the control registers may be easily referenced as long as the appropriate files are included.

3.2 Registers

The ρ-VEX processor has five distinguishable register files. Each is described below.

3.2.1 General purpose registers

The ρ-VEX core contains 64 32-bit general purpose registers for arithmetic.

Register 0 is special, as it always reads as 0 when used by the processor. Writing to it does however work; the debug bus can read the latest value written to it. This allows the register to be used for debugging on rare occasions.

Register 1 is intended to be used as the stack pointer. The `RETURN` and `RFI` instructions can add an immediate value to it for stack adjustment, but otherwise it behaves just as any other general purpose register.

Register 63 can optionally be mapped to the link register at design time using generics. This allows arithmetic instructions to be performed on the link register without needing to use `MOVFL` and `MOVTL`, at the cost of a general purpose register.

There are no explicit move or load-immediate operations, as the following syllables are already capable of these operations.

```
c0 or $r0.dest = $r0.0, $r0.src        // Move src to dest
c0 or $r0.dest = $r0.0, immediate      // Load immediate
```

3.2.2 Branch registers

The ρ-VEX core contains 8 1-bit registers used for branch conditions, select instructions, divisions, and additions of values wider than 32 bits.

All arithmetic operations that output a boolean value can write to either a general purpose register (in which case they will write 0 for false and 1 for true) or a branch register. These include all integer comparison operations and select boolean operations.

Moving a branch register to another branch register cannot be done in a single cycle, but loading an immediate into a branch register or moving to or from a general purpose register can be done as follows.

```
c0 cmpeq $b0.dest = $r0.0, $r0.0       // Load true
c0 cmpne $b0.dest = $r0.0, $r0.0       // Load false
c0 cmpne $b0.dest = $r0.0, $r0.src     // Move general purpose to branch
c0 slctf $r0.dest = $b0.src, $r0.0, 1  // Move branch to general purpose
```

Branch register can also not be loaded from or stored into memory on their own. However, to improve context switching speed slightly, the `LDBR` and `STBR` instructions are available. These load or store a byte containing all eight branch registers in a single syllable.

3.2.3 Link register

The link register is a 32-bit register used to store the return address when calling. It can also be used as the destination address for an unconditional indirect jump or call, in cases where the branch offset field is too small or when the jump target is determined at runtime.

When general purpose register 63 is not mapped to the link register, the `MOVTL` and `LDW` instructions can be used to load the link register from a general purpose register or memory respectively. `MOVFL` and `STW` perform the reverse operations.

3.2.4 Global and context control registers

These two register files contain special-purpose registers. The global control registers contain status information not specific to any context, whereas the context control registers are context specific.

The processor can access these register files through memory operations only. All these accesses are single-cycle. 1 kiB of memory space has to be reserved for this purpose, usually mapped to `0xFFFFFC00..0xFFFFFFFF`. The location of the block is design-time configurable. Note that it is impossible for the processor to perform actual memory operations to this region, so the location of the block should be chosen wisely.

The global register file is read-only from the perspective of the program. The context register file is writable, but it should be noted that each program can only access its own hardware context register file. If an application requires that programs can write to the global register file or the other context register files, the debug bus can be made accessible for memory operations by the bus interconnect outside the core. In most platforms this happens coincidentally, as the processor can access the main bus of the platform, and the debug bus is wired as a slave peripheral on this bus. For more information about the debug bus, refer to Section 9.2.2.7.

For more information about the control registers in general, refer to Section 4.

3.3 Memory

Each lane group of the ρ-VEX processor currently has exactly one memory unit. The configurability of this may be extended in the future, as memory operations commonly end up being the critical path when extracting instruction-level parallelism. However,

doing so would require significant modifications to the ρ-VEX core and the reconfigurable cache.

The ρ-VEX processor is big endian. This means that when accessing a 32-bit or 16-bit word, the most significant byte will reside in the lowest address. This is the opposite of what you may be used to coming from x86.

The ρ-VEX processor is capable of reading and writing 32-bit, 16-bit and 8-bit words. Seperate read instructions exist for reading 16-bit and 8-bit words in signed or unsigned mode. All n-bit accesses must be n-bit aligned. If an access is improperly aligned, a `MISALIGNED_ACCESS` trap will be caused.

Note that a 1 kiB block of the external memory space must be selected to be remapped to the control register file internally. This prevents the processor from being able to access the block. Refer to Section 3.2.4 for more information.

3.4 Syllable resource classes and delays

Some syllables take more than one cycle to complete. In this case, they are always pipelined; no multi-cycle syllable will stall the rest of the bundle. While this is good for performance, it does require the attention of the programmer in order to write properly functioning code.

In addition, not all pipelanes support execution of all syllables, and as such, requirements are imposed on the position of certain syllables within a bundle in the binary. The assembler will normally ensure that these requirements are met, unlike the delay requirements, which it cannot detect. However, it is still important for the user to know them in order to be able to write assembly.

It is also important that the assembler is configured in the same way as the core. If there are discrepancies, the assembler may still output binaries that the core cannot execute. If this happens, the core will produce an `TRAP_INVALID_OP` trap, with the index of the offending pipelane as the trap argument.

There are five distinguishable classes of syllables. These classes are ALU, multiply, memory, branch and long immediate.

3.4.1 ALU class

ALU syllables are the basic ρ-VEX instructions. They can be processed by every lane. In the default pipeline and forwarding configuration of the ρ-VEX, their results are available after a single cycle. That is, the bundle immediately following can use their results.

3.4.2 Multiply class

Multiply syllables are only allowed in lanes that are configured to have a multiplier. This configuration is done at design time using generics. In the default configuration, every lane has a multiplication unit.

In the default pipeline and forwarding configuration of the ρ-VEX, multiply instructions are two-cycle pipelined. That is, two bundle boundaries are needed between the syllable producing the value and a syllable that uses it.

3.4.3 Memory class

Memory syllables are only allowed in lanes that are configured to have a memory unit. In addition, in most configurations, only one memory unit can be active per context at a time, even if multiple are available. This is due to the fact the data cache can only perform one operation per cycle per context. In theory, it is still permissible in such a system to perform a single memory operation and a single control register operation at the same time, but there is currently no toolchain support for this.

In the default pipeline and forwarding configuration of the ρ-VEX, memory load instructions are two-cycle pipelined. That is, two bundle boundaries are needed between a load syllable and the first syllable that uses the value. However, there is no store to load delay; if a bundle with a load of a certain address immediately follows a bundle that stores a value at that address, the newly written value is loaded.

3.4.4 Branch class

All syllables that affect the program counter are considered branch syllables. Only one branch syllable is permitted per cycle, and in almost all design-time core configurations, it must be the last syllable in a bundle.

In the previous ρ-VEX version, a delay was needed between a syllable producing a branch register or link register value and branch operations. This is not the case in the default pipeline and forwarding configuration of this ρ-VEX version, as the ALU and branch operations are initiated in the same pipeline stage.

3.4.5 Long immediate class

Sometimes, one syllable does not contain enough information for one pipelane to execute. The only time when this happens in the ρ-VEX processor is when an immediate outside the range -256..255 is to be specified. For this purpose, `LIMMH` syllables exist. These syllables perform no operation in their own pipelane, but instead send 23 additional immediate bits to another lane, which allows a 32-bit immediate to be used in a single cycle.

Any ALU, multiply or memory syllable that supports an immediate can receive a long immediate. However, long immediates can *not* be used to extend the branch offset field.

`LIMMH` syllables are automatically inferred by the assembler. However, each `LIMMH` syllable inferred means that one less functional syllable can be scheduled in a single bundle. In addition, a certain pipelane can not 'send' a long immediate to any other pipelane.

The ρ-VEX supports two routes for long immediates to take. They are called 'long immediate from neighbor' and 'long immediate from previous pair'. One or both of these methods may be enabled at design time using generics.

Long immediate from neighbor

This is the most common route, as it is supported in all ρ-VEX configurations. This allows all pipelanes to forward a long immediate to their immediate neighbor within a

pair of pipelanes. This is depicted in Figure 3.2 for an 8-way ρ-VEX processor.

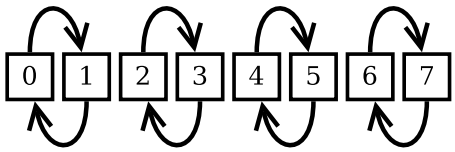


Figure 3.1: Long immediate from neighbor routing.

Long immediate from previous pair

This provides an alternative place where a long immediate can be placed for lanes 2 and up; when this route is enabled lane n can send a long immediate to lane $n + 2$. This is depicted in Figure 3.2 for an 8-way ρ-VEX processor. However, due to limitations in the instruction fetch unit, this system is incompatible with the stop bit system. For these reasons, it can only be effectively used in cores with at least four lanes that are not configured to support stop bits.

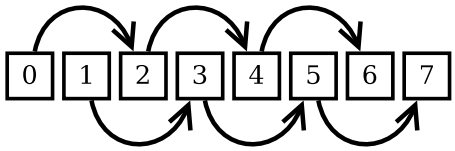


Figure 3.2: Long immediate from previous pair routing.

3.5 Generic binaries

Generic binaries are binaries that can be correctly run on different core configurations, even if the core reconfigures during execution. They were introduced in [2]. Typically, a generic binary refers to a binary that can be run with two pipelanes (2-way), four pipelanes (4-way) or eight pipelanes (8-way).

A generic binary is typically compiled in the same way as a regular 8-way binary. It is the task of the assembler to ensure that the generic binary requirements are met. For the standard generic binary, these rules are the following.

- *The single branch instruction allowed per bundle must end up in the last execution cycle in 2-way and 4-way execution.* The ρ-VEX processor imposes the even stricter requirement that branch syllables must always be the last syllable in a bundle.
- *RAW hazards must be avoided in all runtime configurations.* That is, for example, a register that is written in one of the first two syllables may not be read in subsequent slots. This is because the old value of the register would be read in 8-way mode, but the newly written value would be read in 2-way mode.

Extrapolating these rules to the general case should be trivial.

3.5.1 Generating generic binaries

In order to generate generic binaries, the `-u` flag needs to be passed to the assembler. By default, the assembler will only try to move syllables around within bundles in order to meet the requirements imposed above. However, often this is not possible without further processing.

There are two ways to process the assembly files to meet the requirements. The first one can be done by the assembler as well. If the `-autosplit` flag is passed, it will attempt to split bundles that it cannot schedule directly. This solves most problems at the cost of runtime performance. Refer to [2] for more information.

The second way involves running a python script called `vexparse` on the assembly compilation output, before passing them to the assembler. Depending on its configuration, `vexparse` will extract a dependency graph of all syllables in a basic block[1] from the assembly code, and then completely reschedule all instructions. As a side effect, it will fix hand-written assembly code that failed to take multiply and load instruction delays into consideration.

Being a python script, `vexparse` is much slower than the `-autosplit` option of the assembler. However, it generates more efficient code, as it is not limited to merely splitting bundles.

3.6 Stop bits

The stop bit system is the colloquial name for the binary compression algorithm that the core may be design-time configured to support. It refers to a bit present in every syllable, which, if set, marks the syllable as the last syllable in the current bundle. In contrast, when the stop bit system is not used, bundle boundaries are based on alignment; each bundle is expected to start on an alignment boundary of the maximum size of a bundle. `NOP` instructions are then used to fill the unused words. The stop bit should then still be set in the last syllable, as failing to do so will cause a trap if the bundle contains a branch syllable.

The major advantage of stop bits is the decreased size of the binary. This does not only mean that the memory footprint of a program will be smaller; memory is cheap, so this is usually not an issue. More importantly, it means that the processor will need to do less instruction memory accesses for the same amount of computation; memory bandwidth and caches *are* expensive.

There is an additional benefit when combined with generic binaries. When a generic binary without stop bits runs in 8-way mode, the `NOP` instructions needed for bundle alignment do not cause any delays in execution, aside from the implicit delays due to the strain on the instruction memory system. However, when the binary is run in 2-way mode, these alignment `NOP`s may actually cost cycles. To illustrate, imagine an 8-way generic binary bundle with only two syllables used. When this bundle is executed in 2-way mode, execution will necessarily still take four cycles, because the processor still

[1] A basic block is a block of instructions with natural scheduling boundaries at the start and end of it. The prime example of such boundaries are branch instructions.

needs to work through eight syllables.[2] When stop bits are enabled, such alignment NOPs do not exist, so they will naturally never waste cycles.

The major disadvantage of using stop bits is its hardware complexity. Without stop bits, the core naturally always fetches a nicely aligned block of instruction memory to process. Each 32-bit word in this block can be wired directly to the syllable input of each lane. In contrast, when stop bits are fully enabled, a bundle may start on any 32-bit word boundary. Thus, a new module is needed between the instruction memory (which expects accesses aligned to its access size) and the pipelanes. This module must then be capable of routing any incoming 32-bit word to any pipelane, based on the lower bits of the current program counter and even the syllable type, as branch syllables always need to be routed to the last pipelane. It must also store the previous fetch to handle misaligned bundles, and when a branch to a misaligned address occurs, it must stall execution for an additional cycle, as it will have to fetch both the memory block before and after the crossed alignment boundary.

On the plus side, the large multiplexers involved in this instruction buffer do not increase in size when adding reconfiguration capabilities to an 8-way core with stop bits. Some additional control logic is obviously required, but nothing more.

3.6.1 Design-time configuration

The ρ-VEX processor core allows the designer to make a compromise between the large binary size without stop bits and the additional hardware needed with stop bits. Instead of simply supporting stop bits or not, the stop bit system is configured by specifying the bundle alignment boundaries that the core may expect. When the bundle alignment boundaries equal the size of the maximum bundle size, stop bits are effectively disabled. When the alignment boundary is set to 32-bit words, stop bits are fully enabled. Midway configuration are supported equally well.

Every time the bundle alignment boundary is halved, the multiplexers in the syllable dispatch logic double in size. The complexity of program counter generation increases with each step as well, as does the instruction fetch buffer size. Meanwhile, the number of alignment NOPs required in the binary decreases with each step.

The default 8-way reconfigurable core with stop bits enabled have the bundle alignment boundary set to 64-bit. Going all the way to 32-bit boundaries does not increase 2-way execution performance of an 8-way generic binary further, and most NOPs have already been eliminated, so doubling the hardware complexity once more is generally not justifiable.

[2] It is certainly possible to avoid this without a complete stop bit system. For example, for the previous version of the ρ-VEX processor, it was proposed to use the stop bits to mark the end of the useful part of a bundle, instead of the actual boundaries. In the case of our 8-way bundle with only two syllables used, assuming the two syllables can be placed in the first two slots, the stop bit would be set in the second syllable instead of the eighth. When this code is executed in 2-way mode, the ρ-VEX processor would recognize that it can jump to the next 8-way bundle alignment boundary, thus skipping the six NOP syllables.

3.7 Instruction set

The ρ-VEX instruction set consists of 189 instructions. These instructions are defined by two bitfields in the syllable, called `opcode` and `imm_sw`. The `opcode` field is 8 bits in size, ranging from bit 31 to 24 inclusive, allowing for 256 different operations to be performed. `imm_sw` is a single bit (bit 23) that specifies if the second operand is a register or an immediate. This thus allows a total of 512 different instructions in theory.

However, not all operations support both register and immediate mode. In addition, some instructions have operand fields that extend into the `opcode`, requiring a single instruction to use multiple opcodes. Taking these things into consideration, the ρ-VEX instruction set has 102 `opcode`s that are not yet mapped.

There are two additional fields with a fixed function within the instruction set. The first is the stop bit, bit 1. This bit determines where the bundle boundaries are. Refer to Section 3.6 for more information. The second field, bit 0, is reserved for cluster end bits. The toolchain currently always outputs a 0 bit, and the processor ignores it completely.

The following table lists all the instructions in the ρ-VEX instruction set ordered by opcode. The subsequent sections document each instruction, ordered by function. If you are reading this document digitally, you can click any instruction in the table to jump to its documentation.

31 30 29 28 27 26 25 24	23	22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2	1	0	
0 0 0 0 0 0 0 0	0	d	x	y	S		mpyll $r0.d = $r0.x, $r0.y
0 0 0 0 0 0 0 0	1	d	x	imm	S		mpyll $r0.d = $r0.x, imm
0 0 0 0 0 0 0 1	0	d	x	y	S		mpyllu $r0.d = $r0.x, $r0.y
0 0 0 0 0 0 0 1	1	d	x	imm	S		mpyllu $r0.d = $r0.x, imm
0 0 0 0 0 0 1 0	0	d	x	y	S		mpylh $r0.d = $r0.x, $r0.y
0 0 0 0 0 0 1 0	1	d	x	imm	S		mpylh $r0.d = $r0.x, imm
0 0 0 0 0 0 1 1	0	d	x	y	S		mpylhu $r0.d = $r0.x, $r0.y
0 0 0 0 0 0 1 1	1	d	x	imm	S		mpylhu $r0.d = $r0.x, imm
0 0 0 0 0 1 0 0	0	d	x	y	S		mpyhh $r0.d = $r0.x, $r0.y
0 0 0 0 0 1 0 0	1	d	x	imm	S		mpyhh $r0.d = $r0.x, imm
0 0 0 0 0 1 0 1	0	d	x	y	S		mpyhhu $r0.d = $r0.x, $r0.y
0 0 0 0 0 1 0 1	1	d	x	imm	S		mpyhhu $r0.d = $r0.x, imm
0 0 0 0 0 1 1 0	0	d	x	y	S		mpyl $r0.d = $r0.x, $r0.y
0 0 0 0 0 1 1 0	1	d	x	imm	S		mpyl $r0.d = $r0.x, imm
0 0 0 0 0 1 1 1	0	d	x	y	S		mpylu $r0.d = $r0.x, $r0.y
0 0 0 0 0 1 1 1	1	d	x	imm	S		mpylu $r0.d = $r0.x, imm
0 0 0 0 1 0 0 0	0	d	x	y	S		mpyh $r0.d = $r0.x, $r0.y
0 0 0 0 1 0 0 0	1	d	x	imm	S		mpyh $r0.d = $r0.x, imm
0 0 0 0 1 0 0 1	0	d	x	y	S		mpyhu $r0.d = $r0.x, $r0.y
0 0 0 0 1 0 0 1	1	d	x	imm	S		mpyhu $r0.d = $r0.x, imm
0 0 0 0 1 0 1 0	0	d	x	y	S		mpyhs $r0.d = $r0.x, $r0.y
0 0 0 0 1 0 1 0	1	d	x	imm	S		mpyhs $r0.d = $r0.x, imm
0 0 0 0 1 0 1 1	0			y	S		movtl $l0.0 = $r0.y
0 0 0 0 1 0 1 1	1			imm	S		movtl $l0.0 = imm
0 0 0 0 1 1 0 0	0	d			S		movfl $r0.d = $l0.0
0 0 0 0 1 1 0 1	1		x	imm	S		ldw $l0.0 = imm[$r0.x]
0 0 0 0 1 1 1 0	1		x	imm	S		stw imm[$r0.x] = $l0.0
0 0 0 1 0 0 0 0	1	d	x	imm	S		ldw $r0.d = imm[$r0.x]
0 0 0 1 0 0 0 1	1	d	x	imm	S		ldh $r0.d = imm[$r0.x]
0 0 0 1 0 0 1 0	1	d	x	imm	S		ldhu $r0.d = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	0	1	1	1	d							x								imm						S		ldb $r0.d = imm[$r0.x]
0	0	0	1	0	1	0	0	1	d							x								imm						S		ldbu $r0.d = imm[$r0.x]
0	0	0	1	0	1	0	1	1	d							x								imm						S		stw imm[$r0.x] = $r0.d
0	0	0	1	0	1	1	0	1	d							x								imm						S		sth imm[$r0.x] = $r0.d
0	0	0	1	0	1	1	1	1	d							x								imm						S		stb imm[$r0.x] = $r0.d
0	0	0	1	1	0	0	0	0	d							x								y						S		shr $r0.d = $r0.x, $r0.y
0	0	0	1	1	0	0	0	1	d							x								imm						S		shr $r0.d = $r0.x, imm
0	0	0	1	1	0	0	1	0	d							x								y						S		shru $r0.d = $r0.x, $r0.y
0	0	0	1	1	0	0	1	1	d							x								imm						S		shru $r0.d = $r0.x, imm
0	0	0	1	1	0	1	0	0	d							x								y						S		sub $r0.d = $r0.y, $r0.x
0	0	0	1	1	0	1	0	1	d							x								imm						S		sub $r0.d = imm, $r0.x
0	0	0	1	1	0	1	1	0	d							x														S		sxtb $r0.d = $r0.x
0	0	0	1	1	1	0	0	0	d							x														S		sxth $r0.d = $r0.x
0	0	0	1	1	1	0	1	0	d							x														S		zxtb $r0.d = $r0.x
0	0	0	1	1	1	1	0	0	d							x														S		zxth $r0.d = $r0.x
0	0	0	1	1	1	1	1	0	d							x								y						S		xor $r0.d = $r0.x, $r0.y
0	0	0	1	1	1	1	1	1	d							x								imm						S		xor $r0.d = $r0.x, imm
0	0	1	0	0	0	0	0	0	offs																					S		goto offs
0	0	1	0	0	0	0	0	1																						S		igoto $l0.0
0	0	1	0	0	0	0	1	0	offs																					S		call $l0.0 = offs
0	0	1	0	0	0	0	1	1																						S		icall $l0.0 = $l0.0
0	0	1	0	0	0	1	0	0	offs																			bs		S		br $b0.bs, offs
0	0	1	0	0	0	1	0	1	offs																			bs		S		brf $b0.bs, offs
0	0	1	0	0	0	1	1	0	stackadj																					S		return $r0.1 = $r0.1, stackadj, $l0.0
0	0	1	0	0	0	1	1	1	stackadj																					S		rfi $r0.1 = $r0.1, stackadj
0	0	1	0	1	0	0	0	0																						S		stop
0	0	1	0	1	1	0	0	0	d							x								y						S		sbit $r0.d = $r0.x, $r0.y
0	0	1	0	1	1	0	0	1	d							x								imm						S		sbit $r0.d = $r0.x, imm
0	0	1	0	1	1	0	1	0	d							x								y						S		sbitf $r0.d = $r0.x, $r0.y
0	0	1	0	1	1	0	1	1	d							x								imm						S		sbitf $r0.d = $r0.x, imm
0	0	1	0	1	1	1	0	1								x								imm						S		ldbr imm[$r0.x]
0	0	1	0	1	1	1	1	1								x								imm						S		stbr imm[$r0.x]
0	0	1	1	0	bs			0	d							x								y						S		slctf $r0.d = $b0.bs, $r0.x, $r0.y
0	0	1	1	0	bs			1	d							x								imm						S		slctf $r0.d = $b0.bs, $r0.x, imm
0	0	1	1	1	bs			0	d							x								y						S		slct $r0.d = $b0.bs, $r0.x, $r0.y
0	0	1	1	1	bs			1	d							x								imm						S		slct $r0.d = $b0.bs, $r0.x, imm
0	1	0	0	0	0	0	0	0	d							x								y						S		cmpeq $r0.d = $r0.x, $r0.y
0	1	0	0	0	0	0	0	1	d							x								imm						S		cmpeq $r0.d = $r0.x, imm
0	1	0	0	0	0	0	0	1	0			bd				x								y						S		cmpeq $b0.bd = $r0.x, $r0.y
0	1	0	0	0	0	0	0	1	1			bd				x								imm						S		cmpeq $b0.bd = $r0.x, imm
0	1	0	0	0	0	0	1	0	d							x								y						S		cmpge $r0.d = $r0.x, $r0.y
0	1	0	0	0	0	0	1	0	d							x								imm						S		cmpge $r0.d = $r0.x, imm
0	1	0	0	0	0	0	1	1	0			bd				x								y						S		cmpge $b0.bd = $r0.x, $r0.y
0	1	0	0	0	0	0	1	1	1			bd				x								imm						S		cmpge $b0.bd = $r0.x, imm
0	1	0	0	0	0	1	0	0	d							x								y						S		cmpgeu $r0.d = $r0.x, $r0.y
0	1	0	0	0	0	1	0	0	d							x								imm						S		cmpgeu $r0.d = $r0.x, imm
0	1	0	0	0	0	1	0	1	0			bd				x								y						S		cmpgeu $b0.bd = $r0.x, $r0.y
0	1	0	0	0	0	1	0	1	1			bd				x								imm						S		cmpgeu $b0.bd = $r0.x, imm
0	1	0	0	0	0	1	1	0	d							x								y						S		cmpgt $r0.d = $r0.x, $r0.y
0	1	0	0	0	0	1	1	0	d							x								imm						S		cmpgt $r0.d = $r0.x, imm
0	1	0	0	0	0	1	1	1	0			bd				x								y						S		cmpgt $b0.bd = $r0.x, $r0.y
0	1	0	0	0	0	1	1	1	1			bd				x								imm						S		cmpgt $b0.bd = $r0.x, imm
0	1	0	0	1	0	0	0	0	d							x								y						S		cmpgtu $r0.d = $r0.x, $r0.y
0	1	0	0	1	0	0	0	1	d							x								imm						S		cmpgtu $r0.d = $r0.x, imm

31 30 29 28 27 26 25 24	23	22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2	1	0	
0 1 0 0 1 0 0 1	0	bd	x	y	S		cmpgtu $b0.bd = $r0.x, $r0.y
0 1 0 0 1 0 0 1	1	bd	x	imm	S		cmpgtu $b0.bd = $r0.x, imm
0 1 0 0 1 0 1 0	0	d	x	y	S		cmple $r0.d = $r0.x, $r0.y
0 1 0 0 1 0 1 0	1	d	x	imm	S		cmple $r0.d = $r0.x, imm
0 1 0 0 1 0 1 1	0	bd	x	y	S		cmple $b0.bd = $r0.x, $r0.y
0 1 0 0 1 0 1 1	1	bd	x	imm	S		cmple $b0.bd = $r0.x, imm
0 1 0 0 1 1 0 0	0	d	x	y	S		cmpleu $r0.d = $r0.x, $r0.y
0 1 0 0 1 1 0 0	1	d	x	imm	S		cmpleu $r0.d = $r0.x, imm
0 1 0 0 1 1 0 1	0	bd	x	y	S		cmpleu $b0.bd = $r0.x, $r0.y
0 1 0 0 1 1 0 1	1	bd	x	imm	S		cmpleu $b0.bd = $r0.x, imm
0 1 0 0 1 1 1 0	0	d	x	y	S		cmplt $r0.d = $r0.x, $r0.y
0 1 0 0 1 1 1 0	1	d	x	imm	S		cmplt $r0.d = $r0.x, imm
0 1 0 0 1 1 1 1	0	bd	x	y	S		cmplt $b0.bd = $r0.x, $r0.y
0 1 0 0 1 1 1 1	1	bd	x	imm	S		cmplt $b0.bd = $r0.x, imm
0 1 0 1 0 0 0 0	0	d	x	y	S		cmpltu $r0.d = $r0.x, $r0.y
0 1 0 1 0 0 0 0	1	d	x	imm	S		cmpltu $r0.d = $r0.x, imm
0 1 0 1 0 0 0 1	0	bd	x	y	S		cmpltu $b0.bd = $r0.x, $r0.y
0 1 0 1 0 0 0 1	1	bd	x	imm	S		cmpltu $b0.bd = $r0.x, imm
0 1 0 1 0 0 1 0	0	d	x	y	S		cmpne $r0.d = $r0.x, $r0.y
0 1 0 1 0 0 1 0	1	d	x	imm	S		cmpne $r0.d = $r0.x, imm
0 1 0 1 0 0 1 1	0	bd	x	y	S		cmpne $b0.bd = $r0.x, $r0.y
0 1 0 1 0 0 1 1	1	bd	x	imm	S		cmpne $b0.bd = $r0.x, imm
0 1 0 1 0 1 0 0	0	d	x	y	S		nandl $r0.d = $r0.x, $r0.y
0 1 0 1 0 1 0 0	1	d	x	imm	S		nandl $r0.d = $r0.x, imm
0 1 0 1 0 1 0 1	0	bd	x	y	S		nandl $b0.bd = $r0.x, $r0.y
0 1 0 1 0 1 0 1	1	bd	x	imm	S		nandl $b0.bd = $r0.x, imm
0 1 0 1 0 1 1 0	0	d	x	y	S		norl $r0.d = $r0.x, $r0.y
0 1 0 1 0 1 1 0	1	d	x	imm	S		norl $r0.d = $r0.x, imm
0 1 0 1 0 1 1 1	0	bd	x	y	S		norl $b0.bd = $r0.x, $r0.y
0 1 0 1 0 1 1 1	1	bd	x	imm	S		norl $b0.bd = $r0.x, imm
0 1 0 1 1 0 0 0	0	d	x	y	S		orl $r0.d = $r0.x, $r0.y
0 1 0 1 1 0 0 0	1	d	x	imm	S		orl $r0.d = $r0.x, imm
0 1 0 1 1 0 0 1	0	bd	x	y	S		orl $b0.bd = $r0.x, $r0.y
0 1 0 1 1 0 0 1	1	bd	x	imm	S		orl $b0.bd = $r0.x, imm
0 1 0 1 1 0 1 0	0	d	x	y	S		andl $r0.d = $r0.x, $r0.y
0 1 0 1 1 0 1 0	1	d	x	imm	S		andl $r0.d = $r0.x, imm
0 1 0 1 1 0 1 1	0	bd	x	y	S		andl $b0.bd = $r0.x, $r0.y
0 1 0 1 1 0 1 1	1	bd	x	imm	S		andl $b0.bd = $r0.x, imm
0 1 0 1 1 1 0 0	0	d	x	y	S		tbit $r0.d = $r0.x, $r0.y
0 1 0 1 1 1 0 0	1	d	x	imm	S		tbit $r0.d = $r0.x, imm
0 1 0 1 1 1 0 1	0	bd	x	y	S		tbit $b0.bd = $r0.x, $r0.y
0 1 0 1 1 1 0 1	1	bd	x	imm	S		tbit $b0.bd = $r0.x, imm
0 1 0 1 1 1 1 0	0	d	x	y	S		tbitf $r0.d = $r0.x, $r0.y
0 1 0 1 1 1 1 0	1	d	x	imm	S		tbitf $r0.d = $r0.x, imm
0 1 0 1 1 1 1 1	0	bd	x	y	S		tbitf $b0.bd = $r0.x, $r0.y
0 1 0 1 1 1 1 1	1	bd	x	imm	S		tbitf $b0.bd = $r0.x, imm
0 1 1 0 0 0 0 0					S		nop
0 1 1 0 0 0 0 1 0	0	d	x	y	S		add $r0.d = $r0.x, $r0.y
0 1 1 0 0 0 0 1 0	1	d	x	imm	S		add $r0.d = $r0.x, imm
0 1 1 0 0 0 0 1 1	0	d	x	y	S		and $r0.d = $r0.x, $r0.y
0 1 1 0 0 0 0 1 1	1	d	x	imm	S		and $r0.d = $r0.x, imm
0 1 1 0 0 1 0 0	0	d	x	y	S		andc $r0.d = $r0.x, $r0.y
0 1 1 0 0 1 0 0	1	d	x	imm	S		andc $r0.d = $r0.x, imm
0 1 1 0 0 1 0 1	0	d	x	y	S		max $r0.d = $r0.x, $r0.y

31 30 29 28 27 26 25 24	23	22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2	1	0	
0 1 1 0 0 1 0 1	1	d	x	imm	S		max $r0.d = $r0.x, imm
0 1 1 0 0 1 1 0	0	d	x	y	S		maxu $r0.d = $r0.x, $r0.y
0 1 1 0 0 1 1 0	1	d	x	imm	S		maxu $r0.d = $r0.x, imm
0 1 1 0 0 1 1 1	0	d	x	y	S		min $r0.d = $r0.x, $r0.y
0 1 1 0 0 1 1 1	1	d	x	imm	S		min $r0.d = $r0.x, imm
0 1 1 0 1 0 0 0	0	d	x	y	S		minu $r0.d = $r0.x, $r0.y
0 1 1 0 1 0 0 0	1	d	x	imm	S		minu $r0.d = $r0.x, imm
0 1 1 0 1 0 0 1	0	d	x	y	S		or $r0.d = $r0.x, $r0.y
0 1 1 0 1 0 0 1	1	d	x	imm	S		or $r0.d = $r0.x, imm
0 1 1 0 1 0 1 0	0	d	x	y	S		orc $r0.d = $r0.x, $r0.y
0 1 1 0 1 0 1 0	1	d	x	imm	S		orc $r0.d = $r0.x, imm
0 1 1 0 1 0 1 1	0	d	x	y	S		sh1add $r0.d = $r0.x, $r0.y
0 1 1 0 1 0 1 1	1	d	x	imm	S		sh1add $r0.d = $r0.x, imm
0 1 1 0 1 1 0 0	0	d	x	y	S		sh2add $r0.d = $r0.x, $r0.y
0 1 1 0 1 1 0 0	1	d	x	imm	S		sh2add $r0.d = $r0.x, imm
0 1 1 0 1 1 0 1	0	d	x	y	S		sh3add $r0.d = $r0.x, $r0.y
0 1 1 0 1 1 0 1	1	d	x	imm	S		sh3add $r0.d = $r0.x, imm
0 1 1 0 1 1 1 0	0	d	x	y	S		sh4add $r0.d = $r0.x, $r0.y
0 1 1 0 1 1 1 0	1	d	x	imm	S		sh4add $r0.d = $r0.x, imm
0 1 1 0 1 1 1 1	0	d	x	y	S		shl $r0.d = $r0.x, $r0.y
0 1 1 0 1 1 1 1	1	d	x	imm	S		shl $r0.d = $r0.x, imm
0 1 1 1 0 bs	0	d	x	y bd	S		divs $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y
0 1 1 1 1 bs	0	d	x	y bd	S		addcg $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y
1 0 0 0 tgt	imm				S		limmh tgt, imm
1 0 0 1 0 0 0 0 0	0		x	y	S		trap $r0.x, $r0.y
1 0 0 1 0 0 0 0 0	1		x	imm	S		trap $r0.x, imm
1 0 0 1 0 0 0 1	0	d	x		S		clz $r0.d = $r0.x
1 0 0 1 0 0 1 0	0	d	x	y	S		mpylhus $r0.d = $r0.x, $r0.y
1 0 0 1 0 0 1 0	1	d	x	imm	S		mpylhus $r0.d = $r0.x, imm
1 0 0 1 0 0 1 1	0	d	x	y	S		mpyhhs $r0.d = $r0.x, $r0.y
1 0 0 1 0 0 1 1	1	d	x	imm	S		mpyhhs $r0.d = $r0.x, imm
1 0 0 1 0 1 0 1		d	x		S		convif $r0.d = $r0.x
1 0 0 1 0 1 1 0		d	x		S		convfi $r0.d = $r0.x
1 0 0 1 0 1 1 1	0	d	x	y	S		addf $r0.d = $r0.x, $r0.y
1 0 0 1 0 1 1 1	1	d	x	imm	S		addf $r0.d = $r0.x, imm
1 0 0 1 1 0 0 0	0	d	x	y	S		subf $r0.d = $r0.x, $r0.y
1 0 0 1 1 0 0 0	1	d	x	imm	S		subf $r0.d = $r0.x, imm
1 0 0 1 1 0 0 1	0	d	x	y	S		mpyf $r0.d = $r0.x, $r0.y
1 0 0 1 1 0 0 1	1	d	x	imm	S		mpyf $r0.d = $r0.x, imm
1 0 0 1 1 0 1 0	0	d	x	y	S		cmpgef $r0.d = $r0.x, $r0.y
1 0 0 1 1 0 1 0	1	d	x	imm	S		cmpgef $r0.d = $r0.x, imm
1 0 0 1 1 0 1 1	0	bd	x	y	S		cmpgef $b0.bd = $r0.x, $r0.y
1 0 0 1 1 0 1 1	1	bd	x	imm	S		cmpgef $b0.bd = $r0.x, imm
1 0 0 1 1 1 0 0	0	d	x	y	S		cmpeqf $r0.d = $r0.x, $r0.y
1 0 0 1 1 1 0 0	1	d	x	imm	S		cmpeqf $r0.d = $r0.x, imm
1 0 0 1 1 1 0 1	0	bd	x	y	S		cmpeqf $b0.bd = $r0.x, $r0.y
1 0 0 1 1 1 0 1	1	bd	x	imm	S		cmpeqf $b0.bd = $r0.x, imm
1 0 0 1 1 1 1 0	0	d	x	y	S		cmpgtf $r0.d = $r0.x, $r0.y
1 0 0 1 1 1 1 0	1	d	x	imm	S		cmpgtf $r0.d = $r0.x, imm
1 0 0 1 1 1 1 1	0	bd	x	y	S		cmpgtf $b0.bd = $r0.x, $r0.y
1 0 0 1 1 1 1 1	1	bd	x	imm	S		cmpgtf $b0.bd = $r0.x, imm

3.7.1 ALU arithmetic instructions

The ρ-VEX ALU has a 32-bit adder for arithmetic. Some exotic instructions are available to make efficient multiplications by small constants and to speed up software divisions.

add $r0.d = $r0.x, $r0.y

add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	0	0	d							x					y									S	
0	1	1	0	0	0	1	0	1	d							x					imm									S	

Performs a 32-bit addition. Notice that `ADD` instructions may be used as move or load immediate operations when `x` is set to 0. While the `OR` instruction is often used instead, there is no functional difference between the two when used inthis way.

```
$r0.d = $r0.x + [$r0.y|imm];
```

sh1add $r0.d = $r0.x, $r0.y

sh1add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	1	0	d							x					y									S	
0	1	1	0	1	0	1	1	1	d							x					imm									S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by one.

```
$r0.d = ($r0.x << 1) + [$r0.y|imm];
```

sh2add $r0.d = $r0.x, $r0.y

sh2add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	0	0	d							x					y									S	
0	1	1	0	1	1	0	0	1	d							x					imm									S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by two.

```
$r0.d = ($r0.x << 2) + [$r0.y|imm];
```

sh3add $r0.d = $r0.x, $r0.y

sh3add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	1	0	d							x					y									S	
0	1	1	0	1	1	0	1	1	d							x					imm									S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by three.

```
$r0.d = ($r0.x << 3) + [$r0.y|imm];
```

sh4add $r0.d = $r0.x, $r0.y

sh4add $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	0	0	d							x								y						S	
0	1	1	0	1	1	1	0	1	d							x								imm						S	

Performs a 32-bit addition. `$r0.x` is first left-shifted by four.

```
$r0.d = ($r0.x << 4) + [$r0.y|imm];
```

sub $r0.d = $r0.y, $r0.x

sub $r0.d = imm, $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	0	d							x								y						S	
0	0	0	1	1	0	1	0	1	d							x								imm						S	

Performs a 32-bit subtraction. Note that, unlike all other instructions, the immediate must be specified first. This allows `SUB` to be used to subtract a register from an immediate.

Notice that `SUB` reduces to two's complement negation when `x` or `imm` equal zero.

```
$r0.d = [$r0.y|imm] + $r0.x;
```

addcg $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	bs			0	d							x								y					bd	S	

Primitive for additions of integers wider than 32 bits. Addition is performed by first setting a scratch branch register to false using `CMPNE` for the carry input. Then `ADDCG` can be used to add up words together one by one with increasing significance, using the scratch branch register for the carry chain.

Subtractions can be performed by setting the carry input to 1 using `CMPEQ` and ones-complementing one of the inputs using `XOR`.

Notice that `ADDCG` reduces to rotate left by one through a branch register when `x` equals `y`. This may be used for for shift left by one operations on integers wider than 32 bits.

```
long long tmp = $r0.x + $r0.y + ($b0.bs ? 1 : 0);
$r0.d = (int)tmp;
$b0.bd = (tmp & 0x100000000) != 0;
```

divs $r0.d, $b0.bd = $b0.bs, $r0.x, $r0.y

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	bs			0	d							x								y					bd	S	

Primitive for integer divisions. The following assembly code performs a single nonrestoring division step.

```
addcg  s_lo, shift_bit    = s_lo, s_lo, <zero>
;;
```

```
divs   s_hi, quotient_bit = s_hi, divisor, shift_bit
;;
addcg  quotient, <unused> = quotient, quotient, quotient_bit
;;
```

Here, `s_lo` and `s_hi` represent the partial remainder, initialized to the dividend before the first division step. The first `ADDCG` and the `DIVS` instruction together perform a 64-bit shift-left-by-1 operation. Depending on the sign of the partial remainder before the shift (i.e., the MSB which was shifted out), the dividend is added to or subtracted from the partial remainder. If an addition was performed, `quotient_bit` is set to 1, representing that the current quotient bit is -1 in binary signed digit representation. Otherwise, the current quotient bit is 1. The final `ADDCG` stores the result by left-shifting the bit into `quotient`. Post-processing is required to convert the quotient from binary signed digit representation to two's complement.

It should be noted that a division step can be done in a single cycle using just two syllables. This division step has to be applied many times in a loop, and benefits from modulo-scheduling (also known as software pipelining), allowing each step to be performed in a single cycle. Furthermore, the `quotient_bits` can be shifted into `s_lo` instead of a different register, as the zero bits shifted into `s_lo` are unused. This eliminates the need for the second `ADDCG`, as well as the zero and scratch branch registers.

The prologue and epilogue code for various divisions are beyond the scope of this manual, as the compilers take care of expanding divisions.

```
int tmp = ($r0.x << 1) | ($b0.bs ? 1 : 0);
bool flag = ($r0.x & 0x80000000) != 0;
$r0.d = flag ? (tmp + $r0.y) : (tmp - $r0.y);
$b0.bd = flag;
```

3.7.2 ALU barrel shifter instructions

The ρ-VEX ALU includes a barrel shifter. It should be noted that the shift amount input to the barrel shifter is 8-bit unsigned, not 32-bit as one might expect. That is, the upper 24 bits of the shift amount are discarded, and for instance a left shift by a negative amount will *not* simply result in a right shift.

`shl $r0.d = $r0.x, $r0.y`

`shl $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	1	0	d							x					y									S	
0	1	1	0	1	1	1	1	1	d							x					imm									S	

Performs a left-shift operation. Zeros are shifted in from the right.

```
$r0.d = $r0.x << [$r0.y|imm];
```

shr $r0.d = $r0.x, $r0.y

shr $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	0	d							x					y									S	
0	0	0	1	1	0	0	0	1	d							x					imm									S	

Performs a signed right-shift operation. That is, the sign bit of `$r0.x` is shifted in from the left.

```
$r0.d = $r0.x >> [$r0.y|imm];
```

shru $r0.d = $r0.x, $r0.y

shru $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	1	0	d							x					y									S	
0	0	0	1	1	0	0	1	1	d							x					imm									S	

Performs an unsigned right-shift operation. That is, zeros are shifted in from the left.

```
$r0.d = (unsigned int)$r0.x >> [$r0.y|imm];
```

3.7.3 ALU bitwise instructions

The *ρ*-VEX ALU supports a subset of bitwise operations in a single cycle.

and $r0.d = $r0.x, $r0.y

and $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	1	0	d							x					y									S	
0	1	1	0	0	0	1	1	1	d							x					imm									S	

Performs a bitwise `AND` operation.

```
$r0.d = $r0.x & [$r0.y|imm];
```

andc $r0.d = $r0.x, $r0.y

andc $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	0	d							x					y									S	
0	1	1	0	0	1	0	0	1	d							x					imm									S	

Performs a bitwise `AND` operation, with the first operand one's complemented.

```
$r0.d = ~$r0.x & [$r0.y|imm];
```

or $r0.d = $r0.x, $r0.y

or $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	1	0	d							x					y									S	
0	1	1	0	1	0	0	1	1	d							x					imm									S	

Performs a bitwise `OR` operation. Notice that `OR` instructions reduce to move or load immediate operations when `x` is set to 0.

```
$r0.d = $r0.x | [$r0.y|imm];
```

orc $r0.d = $r0.x, $r0.y

orc $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	0	0	d							x					y									S	
0	1	1	0	1	0	1	0	1	d							x					imm									S	

Performs a bitwise `OR` operation, with the first operand one's complemented. Notice that `ORC` instructions reduce to one's complement when `y` or `imm` is set to 0.

```
$r0.d = ~$r0.x | [$r0.y|imm];
```

xor $r0.d = $r0.x, $r0.y

xor $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	0	d							x					y									S	
0	0	0	1	1	1	1	1	1	d							x					imm									S	

Performs a bitwise `XOR` operation.

```
$r0.d = $r0.x ^ [$r0.y|imm];
```

3.7.4 ALU single-bit instructions

The ρ-VEX ALU supports several bitfield operations in a single cycle. Note that the bit selection logic follows the same rules as the shift amount in the barrel shifter. That is, only the least significant byte of the bit selection operand is used to select the bit, the rest is ignored.

sbit $r0.d = $r0.x, $r0.y

sbit $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	0	d							x					y									S	
0	0	1	0	1	1	0	0	1	d							x					imm									S	

Sets a given bit in a 32-bit integer.

```
$r0.d = $r0.x | (1 << [$r0.y|imm]);
```

sbitf $r0.d = $r0.x, $r0.y

sbitf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	1	0	d							x					y									S	
0	0	1	0	1	1	0	1	1	d							x					imm									S	

Clears a given bit in a 32-bit integer.

```
$r0.d = $r0.x & ~(1 << [$r0.y|imm]);
```

tbit $r0.d = $r0.x, $r0.y

tbit $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	0	0	d							x					y									S	
0	1	0	1	1	1	0	0	1	d							x					imm									S	

Copies a given bit to an integer register.

```
$r0.d = ($r0.x & (1 << [$r0.y|imm])) != 0;
```

tbit $b0.bd = $r0.x, $r0.y

tbit $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	1	0				bd				x					y									S	
0	1	0	1	1	1	0	1	1				bd				x					imm									S	

Copies a given bit to a branch register.

```
$b0.bd = ($r0.x & (1 << [$r0.y|imm])) != 0;
```

tbitf $r0.d = $r0.x, $r0.y

tbitf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	d							x					y									S	
0	1	0	1	1	1	1	0	1	d							x					imm									S	

Copies the complement of a given bit to an integer register.

```
$r0.d = ($r0.x & (1 << [$r0.y|imm])) == 0;
```

tbitf $b0.bd = $r0.x, $r0.y

tbitf $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	1	0				bd				x					y									S	
0	1	0	1	1	1	1	1	1				bd				x					imm									S	

Copies the complement of a given bit to a branch register.

```
$b0.bd = ($r0.x & (1 << [$r0.y|imm])) == 0;
```

3.7.5 ALU boolean instructions

As well as supporting many bitwise operations, the *ρ*-VEX ALU also supports some boolean operations in a single cycle. The boolean operations are defined in the same way

as C boolean operations are defined. That is, the value 0 represents false, and any other value represents true.

andl $r0.d = $r0.x, $r0.y

andl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	0	0	d							x					y									S	
0	1	0	1	1	0	1	0	1	d							x					imm									S	

Performs a boolean AND operation and stores the result in an integer register.

```
$r0.d = $r0.x && [$r0.y|imm];
```

andl $b0.bd = $r0.x, $r0.y

andl $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	1	0				bd				x					y									S	
0	1	0	1	1	0	1	1	1				bd				x					imm									S	

Performs a boolean AND operation and stores the result in a branch register.

```
$b0.bd = $r0.x && [$r0.y|imm];
```

orl $r0.d = $r0.x, $r0.y

orl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	0	0	d							x					y									S	
0	1	0	1	1	0	0	0	1	d							x					imm									S	

Performs a boolean OR operation and stores the result in an integer register.

```
$r0.d = $r0.x || [$r0.y|imm];
```

orl $b0.bd = $r0.x, $r0.y

orl $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	1	0				bd				x					y									S	
0	1	0	1	1	0	0	1	1				bd				x					imm									S	

Performs a boolean OR operation and stores the result in a branch register.

```
$b0.bd = $r0.x || [$r0.y|imm];
```

nandl $r0.d = $r0.x, $r0.y

nandl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	0	d							x					y									S	
0	1	0	1	0	1	0	0	1	d							x					imm									S	

Performs a boolean NAND operation and stores the result in an integer register.

```
$r0.d = !($r0.x && [$r0.y|imm]);
```

nandl $b0.bd = $r0.x, $r0.y

nandl $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	1	0				bd				x					y									S	
0	1	0	1	0	1	0	1	1				bd				x					imm									S	

Performs a boolean NAND operation and stores the result in a branch register.

```
$b0.bd = !($r0.x && [$r0.y|imm]);
```

norl $r0.d = $r0.x, $r0.y

norl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	0	0				d				x					y									S	
0	1	0	1	0	1	1	0	1				d				x					imm									S	

Performs a boolean NOR operation and stores the result in an integer register.

```
$r0.d = !($r0.x || [$r0.y|imm]);
```

norl $b0.bd = $r0.x, $r0.y

norl $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	1	0				bd				x					y									S	
0	1	0	1	0	1	1	1	1				bd				x					imm									S	

Performs a boolean NOR operation and stores the result in a branch register.

```
$b0.bd = !($r0.x || [$r0.y|imm]);
```

3.7.6 ALU compare instructions

The ρ-VEX ALU supports all 32-bit possible integer comparison operations in a single cycle. The immediate version of CMPNE that writes to a branch register is used to load an immediate branch register.

cmpeq $r0.d = $r0.x, $r0.y

cmpeq $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0				d				x					y									S	
0	1	0	0	0	0	0	0	1				d				x					imm									S	

Determines whether the first operand is equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x == [$r0.y|imm];
```

cmpeq $b0.bd = $r0.x, $r0.y

cmpeq $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0				bd				x					y									S	
0	1	0	0	0	0	0	1	1				bd				x					imm									S	

Determines whether the first operand is equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x == [$r0.y|imm];
```

cmpge $r0.d = $r0.x, $r0.y

cmpge $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0				d				x					y									S	
0	1	0	0	0	0	1	0	1				d				x					imm									S	

Determines whether the first operand is greater than or equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x >= [$r0.y|imm];
```

cmpge $b0.bd = $r0.x, $r0.y

cmpge $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0				bd				x					y									S	
0	1	0	0	0	0	1	1	1				bd				x					imm									S	

Determines whether the first operand is greater than or equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x >= [$r0.y|imm];
```

cmpgeu $r0.d = $r0.x, $r0.y

cmpgeu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0				d				x					y									S	
0	1	0	0	0	1	0	0	1				d				x					imm									S	

Determines whether the first operand is greater than or equal to the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x >= (unsigned int)[$r0.y|imm];
```

cmpgeu $b0.bd = $r0.x, $r0.y

cmpgeu $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	0				bd				x					y									S	
0	1	0	0	0	1	0	1	1				bd				x					imm									S	

Determines whether the first operand is greater than or equal to the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x >= (unsigned int)[$r0.y|imm];
```

cmpgt $r0.d = $r0.x, $r0.y

cmpgt $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	0	d							x					y									S	
0	1	0	0	0	1	1	0	1	d							x					imm									S	

Determines whether the first operand is greater than the second operand and stores the result in an integer register.

```
$r0.d = $r0.x > [$r0.y|imm];
```

cmpgt $b0.bd = $r0.x, $r0.y

cmpgt $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0				bd				x					y									S	
0	1	0	0	0	1	1	1	1				bd				x					imm									S	

Determines whether the first operand is greater than the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x > [$r0.y|imm];
```

cmpgtu $r0.d = $r0.x, $r0.y

cmpgtu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	d							x					y									S	
0	1	0	0	1	0	0	0	1	d							x					imm									S	

Determines whether the first operand is greater than the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x > (unsigned int)[$r0.y|imm];
```

cmpgtu $b0.bd = $r0.x, $r0.y

cmpgtu $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	1	0				bd				x					y									S	
0	1	0	0	1	0	0	1	1				bd				x					imm									S	

Determines whether the first operand is greater than the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x > (unsigned int)[$r0.y|imm];
```

cmple $r0.d = $r0.x, $r0.y

cmple $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	0	d							x					y									S	
0	1	0	0	1	0	1	0	1	d							x					imm									S	

Determines whether the first operand is less than or equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x <= [$r0.y|imm];
```

cmple $b0.bd = $r0.x, $r0.y

cmple $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	1	0				bd				x					y									S	
0	1	0	0	1	0	1	1	1				bd				x					imm									S	

Determines whether the first operand is less than or equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x <= [$r0.y|imm];
```

cmpleu $r0.d = $r0.x, $r0.y

cmpleu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	d							x					y									S	
0	1	0	0	1	1	0	0	1	d							x					imm									S	

Determines whether the first operand is less than or equal to the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

cmpleu $b0.bd = $r0.x, $r0.y

cmpleu $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	1	0				bd				x					y									S	
0	1	0	0	1	1	0	1	1				bd				x					imm									S	

Determines whether the first operand is less than or equal to the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

cmplt $r0.d = $r0.x, $r0.y

cmplt $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	d							x					y									S	
0	1	0	0	1	1	1	0	1	d							x					imm									S	

Determines whether the first operand is less than the second operand and stores the result in an integer register.

```
$r0.d = $r0.x <= [$r0.y|imm];
```

cmplt $b0.bd = $r0.x, $r0.y

cmplt $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	1	0				bd				x					y									S	
0	1	0	0	1	1	1	1	1				bd				x					imm									S	

Determines whether the first operand is less than the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x <= [$r0.y|imm];
```

cmpltu $r0.d = $r0.x, $r0.y

cmpltu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	0	0			d					x					y									S	
0	1	0	1	0	0	0	0	1			d					x					imm									S	

Determines whether the first operand is less than the second operand in unsigned arithmetic and stores the result in an integer register.

```
$r0.d = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

cmpltu $b0.bd = $r0.x, $r0.y

cmpltu $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	1	0				bd				x					y									S	
0	1	0	1	0	0	0	1	1				bd				x					imm									S	

Determines whether the first operand is less than the second operand in unsigned arithmetic and stores the result in a branch register.

```
$b0.bd = (unsigned int)$r0.x <= (unsigned int)[$r0.y|imm];
```

cmpne $r0.d = $r0.x, $r0.y

cmpne $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	0	0			d					x					y									S	
0	1	0	1	0	0	1	0	1			d					x					imm									S	

Determines whether the first operand is not equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x != [$r0.y|imm];
```

cmpne $b0.bd = $r0.x, $r0.y

cmpne $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	1	0				bd				x					y									s	
0	1	0	1	0	0	1	1	1				bd				x					imm									s	

Determines whether the first operand is not equal to the second operand and stores the result in a branch register.

Notice that the immediate version of CMPNE reduces to a load immediate operation for branch registers when x is zero.

```
$b0.bd = $r0.x != [$r0.y|imm];
```

3.7.7 ALU selection instructions

The ρ-VEX ALU has single-cycle instructions for conditional moves and computation of the minimum and maximum of two integer values.

slct $r0.d = $b0.bs, $r0.x, $r0.y

slct $r0.d = $b0.bs, $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	bs			0				d				x					y									s	
0	0	1	1	1	bs			1				d				x					imm									s	

Conditional move.

```
$r0.d = $b0.bs ? $r0.x : [$r0.y|imm];
```

slctf $r0.d = $b0.bs, $r0.x, $r0.y

slctf $r0.d = $b0.bs, $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	bs			0				d				x					y									s	
0	0	1	1	0	bs			1				d				x					imm									s	

Conditional move, with operands swapped with respect to SLCT.

Notice that the immediate version of SLCTF reduces to a move from a branch register to an integer register when x is 0 and y is 1.

```
$r0.d = $b0.bs ? [$r0.y|imm] : $r0.x;
```

max $r0.d = $r0.x, $r0.y

max $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	0				d				x					y									s	
0	1	1	0	0	1	0	1	1				d				x					imm									s	

Computes maximum of the input operands using signed arithmetic.

```
$r0.d = ($r0.x >= [$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

maxu $r0.d = $r0.x, $r0.y

maxu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	0	0	d							x					y									S	
0	1	1	0	0	1	1	0	1	d							x					imm									S	

Computes maximum of the input operands using unsigned arithmetic.

```
$r0.d = ((unsigned int)$r0.x >= (unsigned int)[$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

min $r0.d = $r0.x, $r0.y

min $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	1	0	d							x					y									S	
0	1	1	0	0	1	1	1	1	d							x					imm									S	

Computes minimum of the input operands using signed arithmetic.

```
$r0.d = ($r0.x <= [$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

minu $r0.d = $r0.x, $r0.y

minu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	0	0	d							x					y									S	
0	1	1	0	1	0	0	0	1	d							x					imm									S	

Computes minimum of the input operands using unsigned arithmetic.

```
$r0.d = ((unsigned int)$r0.x <= (unsigned int)[$r0.y|imm]) : $r0.x ? [$r0.y|imm];
```

3.7.8 ALU type conversion instructions

The ρ-VEX ALU is capable of supporting typecasts from 32-bit integers to 16-bit and 8-bit integers in a single cycle.

sxtb $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	1	0	d							x														S	

Performs sign extension for an 8-bit value.

```
$r0.d = (char)$r0.x;
```

sxth $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	0	0	d							x														S	

Performs sign extension for a 16-bit value.

```
$r0.d = (short)$r0.x;
```

zxtb $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	0	d							x														S	

Performs zero extension for an 8-bit value.

```
$r0.d = (unsigned char)$r0.x;
```

zxth $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	0	0	d							x														S	

Performs zero extension for a 16-bit value.

```
$r0.d = (unsigned short)$r0.x;
```

3.7.9 ALU miscellaneous instructions

nop

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0																							S	

Performs no operation.

clz $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	1	0	d							x														S	

This operations counts the number of leading zeros in the operand. That is, the value 0x80000000 returns 0 and the value 0 returns 32.

```
unsigned int in = $r0.x;
int out = 32;
while (in) {
  in >>= 1;
  out--;
}
$r0.d = out;
```

movtl $l0.0 = $r0.y

movtl $l0.0 = imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0													y									S	
0	0	0	0	1	0	1	1	1													imm									S	

Copies a general purpose register or immediate to the link register.

```
$l0.0 = [$r0.y|imm];
```

movfl $r0.d = $l0.0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	0	d																					S	

Copies the link register to a general purpose register.

```
$r0.d = $l0.0;
```

trap $r0.x, $r0.y

trap $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0	0								x					y									S	
1	0	0	1	0	0	0	0	1								x					imm									S	

Software trap. The first parameter is the trap argument, while the second parameter is the trap cause byte.

3.7.10 Multiply instructions

ρ-VEX pipelanes may be design-time configured to contain a multiplication unit. This unit supports 16x16 and 16x32 multiplications.

In the default pipeline configuration, these instructions are pipelined by two cycles. That is, the result of a multiply instruction is not available yet in the subsequent instruction.

mpyll $r0.d = $r0.x, $r0.y

mpyll $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	d							x					y									S	
0	0	0	0	0	0	0	0	1	d							x					imm									S	

Signed 16x16-bit to 32-bit multiplication.

```
$r0.d = (short)$r0.x * (short)[$r0.y|imm];
```

mpyllu $r0.d = $r0.x, $r0.y

mpyllu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	d							x					y									S	
0	0	0	0	0	0	0	1	1	d							x					imm									S	

Unsigned 16x16-bit to 32-bit multiplication.

```
$r0.d = (unsigned short)$r0.x * (unsigned short)[$r0.y|imm];
```

mpylh $r0.d = $r0.x, $r0.y

mpylh $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	d							x					y									S	
0	0	0	0	0	0	1	0	1	d							x					imm									S	

Signed 16x16-bit to 32-bit multiplication, using the high halfword of `[$r0.y|imm]`.

```
$r0.d = (short)$r0.x * (short)([$r0.y|imm] >> 16);
```

mpylhu $r0.d = $r0.x, $r0.y

mpylhu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	d							x					y									S	
0	0	0	0	0	0	1	1	1	d							x					imm									S	

Unsigned 16x16-bit to 32-bit multiplication, using the high halfword of `[$r0.y|imm]`.

```
$r0.d = (unsigned short)$r0.x * (unsigned short)([$r0.y|imm] >> 16);
```

mpyhh $r0.d = $r0.x, $r0.y

mpyhh $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	d							x					y									S	
0	0	0	0	0	1	0	0	1	d							x					imm									S	

Signed 16x16-bit to 32-bit multiplication, using the high halfword of both operands.

```
$r0.d = (short)($r0.x >> 16) * (short)([$r0.y|imm] >> 16);
```

mpyhhu $r0.d = $r0.x, $r0.y

mpyhhu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	d							x					y									S	
0	0	0	0	0	1	0	1	1	d							x					imm									S	

Unsigned 16x16-bit to 32-bit multiplication, using the high halfword of both operands.

```
$r0.d = (unsigned short)($r0.x >> 16) * (unsigned short)([$r0.y|imm] >> 16);
```

mpyl $r0.d = $r0.x, $r0.y

mpyl $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	d							x					y									S	
0	0	0	0	0	1	1	0	1	d							x					imm									S	

Signed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

```
$r0.d = (int)$r0.x * (short)[$r0.y|imm];
```

mpylu $r0.d = $r0.x, $r0.y

mpylu $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	d							x					y									S	
0	0	0	0	0	1	1	1	1	d							x					imm									S	

Unsigned 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

This operation may be used as a primitive for 32x32-bit to 32-bit multiplication, computing the partial product of `$r0.x` and the lower half of `[$r0.y|imm]`. `MPYHS` is then used to compute the other partial product. A final `ADD` instruction combines the partial products.

```
$r0.d = (unsigned int)$r0.x * (unsigned short)[$r0.y|imm];
```

`mpyh $r0.d = $r0.x, $r0.y`

`mpyh $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	d							x					y									S	
0	0	0	0	1	0	0	0	1	d							x					imm									S	

Signed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, the upper halfword of `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

```
$r0.d = (int)$r0.x * (short)([$r0.y|imm] >> 16);
```

`mpyhu $r0.d = $r0.x, $r0.y`

`mpyhu $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	d							x					y									S	
0	0	0	0	1	0	0	1	1	d							x					imm									S	

Unsigned 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, the upper halfword of `[$r0.y|imm]` is the 16-bit operand. The upper 16 bits of the multiplication result are discarded.

```
$r0.d = (unsigned int)$r0.x * (unsigned short)([$r0.y|imm] >> 16);
```

`mpyhs $r0.d = $r0.x, $r0.y`

`mpyhs $r0.d = $r0.x, imm`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	d							x					y									S	
0	0	0	0	1	0	1	0	1	d							x					imm									S	

Signed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, the upper halfword of `[$r0.y|imm]` is the 16-bit operand. The result is shifted left by 16, discarding the upper 32 bits of the 48-bit result.

This operation may be used as a primitive for 32x32-bit to 32-bit multiplication, computing the partial product of `$r0.x` and the upper half of `[$r0.y|imm]`. `MPYLU` is then used to compute the other partial product. A final `ADD` instruction combines the partial products.

```
$r0.d = ((int)$r0.x * (short)[$r0.y|imm]) << 16;
```

mpylhus $r0.d = $r0.x, $r0.y

mpylhus $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	0	d							x					y									S	
1	0	0	1	0	0	1	0	1	d							x					imm									S	

Mixed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, interpreted as a signed number. `[$r0.y|imm]` is the 16-bit operand, interpreted as an unsigned number. The 48-bit result is shifted right by 32 bits.

Together with `MPYHS`, `MPYLU`, `MPYHHS`, `ADD` and `ADDCG`, a full signed 32x32-bit to 64-bit multiplication may be realized as follows.

```
mpylu   low1  = op1, op2
mpyhs   low2  = op1, op2
;;
mpylhus high1 = op1, op2
mpyhhs  high2 = op1, op2
;;
addcg   low, carry = <zero>, low1, low2
;;
addcg   high, carry = carry, high1, high2
;;
```

The first two multiply instructions and the first `ADDCG` compute the low word of the multiplication and carry bit for the high word. The remaining instructions do the same for the high part of the result.

```
$r0.d = ((long long)$r0.x * (short)[$r0.y|imm]) >> 32;
```

mpyhhs $r0.d = $r0.x, $r0.y

mpyhhs $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	1	0	d							x					y									S	
1	0	0	1	0	0	1	1	1	d							x					imm									S	

Signed 16x32-bit to 32-bit multiplication. `$r0.x` is the 32-bit operand, the upper halfword of `[$r0.y|imm]` is the 16-bit operand. The 48-bit result is shifted right by 16 bits.

This syllable is used in conjunction with other multiplication syllables for a 32x32-bit to 64-bit signed multiplication. Refer to `MPYLHUS` for more information.

```
$r0.d = ((long long)$r0.x * (short)[$r0.y|imm]) >> 32;
```

3.7.11 Floating-point instructions

addf $r0.d = $r0.x, $r0.y

addf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	1	0	d							x					y									S	
1	0	0	1	0	1	1	1	1	d							x					imm									S	

Performs a single-precision floating-point addition.

```
$r0.d = $r0.x + [$r0.y|imm];
```

subf $r0.d = $r0.x, $r0.y

subf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	0	0	d							x					y									S	
1	0	0	1	1	0	0	0	1	d							x					imm									S	

Performs a single-precision floating-point subtraction.

```
$r0.d = $r0.x - [$r0.y|imm];
```

mpyf $r0.d = $r0.x, $r0.y

mpyf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	0	1	0	d							x					y									S	
1	0	0	1	1	0	0	1	1	d							x					imm									S	

Performs a single-precision floating-point multiplication.

```
$r0.d = $r0.x * [$r0.y|imm];
```

cmpgef $r0.d = $r0.x, $r0.y

cmpgef $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0	0	d							x					y									S	
1	0	0	1	1	0	1	0	1	d							x					imm									S	

Single-precision floating-point compare. Determines whether the first operand is greater than or equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x >= [$r0.y|imm];
```

cmpgef $b0.bd = $r0.x, $r0.y

cmpgef $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1	0				bd				x					y									S	
1	0	0	1	1	0	1	1	1				bd				x					imm									S	

Single-precision floating-point compare. Determines whether the first operand is greater than or equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x >= [$r0.y|imm];
```

cmpeqf $r0.d = $r0.x, $r0.y

cmpeqf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	0	0	d							x					y									S	
1	0	0	1	1	1	0	0	1	d							x					imm									S	

Single-precision floating-point compare. Determines whether the first operand is equal to the second operand and stores the result in an integer register.

```
$r0.d = $r0.x == [$r0.y|imm];
```

cmpeqf $b0.bd = $r0.x, $r0.y

cmpeqf $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	1	0				bd				x					y									S	
1	0	0	1	1	1	0	1	1				bd				x					imm									S	

Single-precision floating-point compare. Determines whether the first operand is equal to the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x == [$r0.y|imm];
```

cmpgtf $r0.d = $r0.x, $r0.y

cmpgtf $r0.d = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	0	0	d							x					y									S	
1	0	0	1	1	1	1	0	1	d							x					imm									S	

Single-precision floating-point compare. Determines whether the first operand is greater than the second operand and stores the result in an integer register.

```
$r0.d = $r0.x > [$r0.y|imm];
```

cmpgtf $b0.bd = $r0.x, $r0.y

cmpgtf $b0.bd = $r0.x, imm

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	1	0				bd				x					y									S	
1	0	0	1	1	1	1	1	1				bd				x					imm									S	

Single-precision floating-point compare. Determines whether the first operand is greater than the second operand and stores the result in a branch register.

```
$b0.bd = $r0.x > [$r0.y|imm];
```

convif $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	1		d							x														S	

Converts a signed integer to a single-precision floating-point number.

```
$r0.d = (float) $r0.x;
```

convfi $r0.d = $r0.x

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	0		d							x														S	

Converts a single-precision floating-point number to a signed integer.

```
$r0.d = (int) $r0.x;
```

3.7.12 Memory instructions

Some ρ-VEX pipelanes have a memory unit. The memory unit supports byte, halfword and word operations. Sign or zero extension is part of the byte and halfword load instructions.

The addressing mode is always register + immediate. Note that attempts to read misaligned memory locations will fail with a `TRAP_MISALIGNED_ACCESS` trap.

In the default pipeline configuration, these instructions are pipelined by two cycles. That is, the result of a memory load instruction is not available yet in the subsequent instruction. However, the current cache and core guarantee that a memory write to address x immediately followed by a memory read from address x returns the newly written value.

ldw $r0.d = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	1		d						x								imm						S	

Loads a 32-bit word from memory.

```
$r0.d = *(int*)($r0.x + imm);
```

ldh $r0.d = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	1		d						x								imm						S	

Loads a 16-bit halfword from memory and sign-extends it.

```
$r0.d = *(short*)($r0.x + imm);
```

ldhu $r0.d = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	0	1		d						x								imm						S	

Loads a 16-bit halfword from memory and zero-extends it.

```
$r0.d = *(unsigned short*)($r0.x + imm);
```

ldb $r0.d = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	1	d							x					imm									S	

Loads a byte from memory and sign-extends it.

```
$r0.d = *(char*)($r0.x + imm);
```

ldbu $r0.d = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	1	d							x					imm									S	

Loads a byte from memory and zero-extends it.

```
$r0.d = *(unsigned char*)($r0.x + imm);
```

ldw $l0.0 = imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1								x					imm									S	

Loads a word from memory. The result is written to the link register.

```
$l0.0 = *(int*)($r0.x + imm);
```

ldbr imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	0	1								x					imm									S	

Loads a byte from memory. The result is written to the entire branch register file at once. This is intended to improve context switching performance somewhat.

```
char tmp = *(char*)($r0.x + imm);
$b0.0 = tmp & 1;
$b0.1 = tmp & 2;
$b0.2 = tmp & 4;
$b0.3 = tmp & 8;
$b0.4 = tmp & 16;
$b0.5 = tmp & 32;
$b0.6 = tmp & 64;
$b0.7 = tmp & 128;
```

stw imm[$r0.x] = $r0.d

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	1	d							x					imm									S	

Stores a 32-bit word into memory.

```
*(int*)($r0.x + imm) = $r0.d;
```

sth imm[$r0.x] = $r0.d

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	0	1	d							x								imm						S	

Stores a 16-bit halfword into memory.

```
*(short*)($r0.x + imm) = $r0.d;
```

stb imm[$r0.x] = $r0.d

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	1	1	1	d							x								imm						S	

Stores a byte into memory.

```
*(char*)($r0.x + imm) = $r0.d;
```

stw imm[$r0.x] = $l0.0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	1								x								imm						S	

Store word in memory, from link register.

```
*(int*)($r0.x + imm) = $l0.0;
```

stbr imm[$r0.x]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	1	1								x								imm						S	

Store byte in memory, from branch register file.

```
char tmp = $b0.0;
tmp |= $b0.1 << 1
tmp |= $b0.2 << 2
tmp |= $b0.3 << 3
tmp |= $b0.4 << 4
tmp |= $b0.5 << 5
tmp |= $b0.6 << 6
tmp |= $b0.7 << 7
*(char*)($r0.x + imm) = tmp
```

3.7.13 Branch instructions

The highest-indexed pipelane in every ρ-VEX system (i.e., the pipelane that the last syllable in a bundle maps to) contains a branch unit. This unit supports the flow control operations outlined below.

Branch offsets are signed immediates relative to the next program counter (`PC+1`). Because there are certain alignment requirements to program counters, the lower two or three bits of the offset are not actually included in the bitfield. Whether this value is two or three depends on the value of the `BRANCH_OFFS_SHIFT` constant defined

in `core_intIface_pkg.vhd`; it is three by default. It must be set to two to support branching to the start of any bundle when stop bits are fully enabled. This must then also be updated in the assembler.

Note that branch offsets and the stack adjust immediate are not eligible for long immediate instructions.

`goto offs`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	offs																						S	

Branches to `PC+1` + `offs` unconditionally.

```
PCP1 += offs;
```

`igoto $l0.0`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	1																							S	

Branches to the address in `$l0.0` unconditionally. This is used for branches to code that cannot be reached using the branch offset immediate.

```
PCP1 = $l0.0;
```

`call $l0.0 = offs`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	0	offs																						S	

Branches to `PC+1` + `offs` unconditionally, while storing `PC+1` in the link register. This is used for function calls.

```
$l0.0 = PCP1;
PCP1 += offs;
```

`icall $l0.0 = $l0.0`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	1	1																							S	

Branches to the address in `$l0.0` unconditionally, while storing `PC+1` in the link register. In other words, it essentially swaps `PC+1` and `$l0.0`. This is used for dynamic function calls or calls to functions that cannot be reached using the branch offset immediate method.

```
unsigned int tmp = $l0.0;
$l0.0 = PCP1;
PCP1 = tmp;
```

`br $b0.bs, offs`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	0	offs																				bs		S	

Branches to `PC+1 + offs` only if `$b0.bs` is true. This instruction performs no operation if `$b0.bs` is false.

```
PCP1 += $b0.bs ? offs : 0;
```

`brf $b0.bs, offs`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	offs																			bs			S	

Branches to `PC+1 + offs` only if `$b0.bs` is false. This instruction performs no operation if `$b0.bs` is true.

```
PCP1 += $b0.bs ? 0 : offs;
```

`return $r0.1 = $r0.1, stackadj, $l0.0`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	0	stackadj																						S	

Returns from a function by branching to `$l0.0` unconditionally, while adding `stackadj` to `$r0.1`. `stackadj` is interpreted as a signed immediate. This allows final stack pointer adjustment and returning to be done with a single syllable.

Notice that this instruction is identical to `IGOTO`, except for the fact that `IGOTO` does not access `$r0.1`.

```
$r0.1 += stackadj;
PCP1 = $l0.0;
```

`rfi $r0.1 = $r0.1, stackadj`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	1	stackadj																						S	

Returns from a trap service routine by branching to `CR_TP` unconditionally and restoring `CR_SCCR` to `CR_CCR`, while adding `stackadj` to `$r0.1`. `stackadj` is interpreted as a signed immediate. This allows final stack pointer adjustment and returning to be done with a single syllable.

```
$r0.1 += stackadj;
CR_CCR = CR_SCCR;
PCP1 = CR_TP;
```

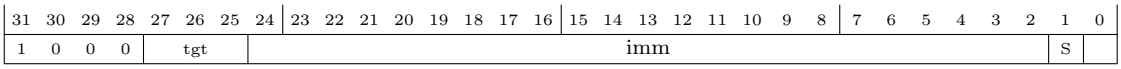
`stop`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	0	0																							S	

Causes a `TRAP_STOP` trap to occur during execution of the next instruction. The `TRAP_STOP` trap will cause the B flag in `CR_DCR` to be set, which will stop execution. Thus, the processor will be stopped after the bundle in which the `STOP` instruction resides is executed.

3.7.14 Long immediate instructions

`limmh tgt, imm`



This special instruction forwards `imm` to lane `tgt`. Actually, only the least significant bit of `tgt` is used by the processor, to distinguish between the two possible long immediate forwarding paths. Refer to Section 3.4.5 for more information.

4 Control registers

The ρ-VEX processor has two control register files. These are the global control register file (`gbreg`) and the context control register file (`cxreg`).

The `gbreg` file contains mostly status information, such as a general purpose cycle counter, the current configuration vector and design-time configuration information. While the debug bus has read/write access to `gbreg`, the core can only read from it.

For more information, refer to Section 3.2.4.

4.1 Global control registers

The following table lists the global control registers of the ρ-VEX processor. The offsets listed are with respect to the control register base address. If you are viewing this manual digitally, you can click the register mnemonics on the right to jump to their documentation.

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x000	R		E B RID		CR_GSR
0x004	BCRR				CR_BCRR
0x008	CC				CR_CC
0x00C	AF				CR_AFF
0x010	CNT				CR_CNT
0x014	CNTH			CNT	CR_CNTH
...	*Unused*				
0x0A0	BORROW15		BORROW14		CR_LIMC7
0x0A4	BORROW13		BORROW12		CR_LIMC6
0x0A8	BORROW11		BORROW10		CR_LIMC5
0x0AC	BORROW9		BORROW8		CR_LIMC4
0x0B0	BORROW7		BORROW6		CR_LIMC3
0x0B4	BORROW5		BORROW4		CR_LIMC2
0x0B8	BORROW3		BORROW2		CR_LIMC1
0x0BC	BORROW1		BORROW0		CR_LIMC0
0x0C0	SYL15CAP	SYL14CAP	SYL13CAP	SYL12CAP	CR_SIC3
0x0C4	SYL11CAP	SYL10CAP	SYL9CAP	SYL8CAP	CR_SIC2
0x0C8	SYL7CAP	SYL6CAP	SYL5CAP	SYL4CAP	CR_SIC1
0x0CC	SYL3CAP	SYL2CAP	SYL1CAP	SYL0CAP	CR_SIC0
0x0D0					CR_GPS1
0x0D4	MEMAR	MEMDC MEMDR	MULC MULR	ALUC ALUR	CR_GPS0
0x0D8					CR_SPS1
0x0DC	MEMMC MEMMR	MEMDC MEMDR	BRC BRR	ALUC ALUR	CR_SPS0
0x0E0					CR_EXT2
0x0E4					CR_EXT1
0x0E8	T BRK	C P		O L F	CR_EXT0
0x0EC			BA NC	NG NL	CR_DCFG
0x0F0	VER	CTAG0	CTAG1	CTAG2	CR_CVER1
0x0F4	CTAG3	CTAG4	CTAG5	CTAG6	CR_CVER0

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0F8	COID	PTAG0	PTAG1	PTAG2	CR_PVER1
0x0FC	PTAG3	PTAG4	PTAG5	PTAG6	CR_PVER0

4.1.1 CR_GSR - Global status register

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x000	R		E B RID		CR_GSR
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core					
Debug	✓				

This register contains miscellaneous status information.

R flag, bit 31

Reset flag. The entire ρ-VEX processor will be reset when the debug bus writes a one to this flag. Writing a zero has no effect.

E flag, bit 13

Reconfiguration error flag. This flag is set by hardware when an invalid configuration was requested. It is cleared once a valid configuration is requested.

B flag, bit 12

Reconfiguration busy flag. While high, reconfiguration requests are ignored.

RID field, bits 11..8

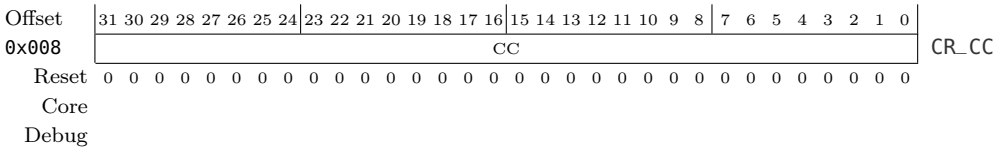
Reconfiguration requester ID. When a configuration is requested, this field is set to the context ID of the context that requested the configuration, or to 0xF if the request was from the debug bus. This may be used by the reconfiguration sources to see if they have won arbitration. Refer to Section 6.2 for more information.

4.1.2 CR_BCRR - Bus reconfiguration request register

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x004	BCRR				CR_BCRR
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core					
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

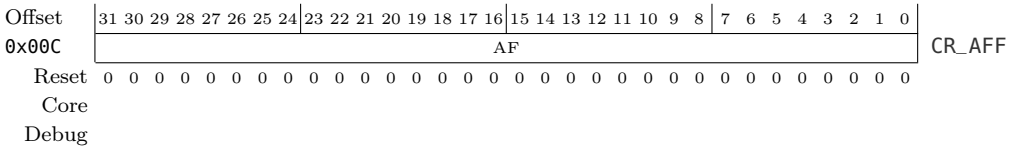
This register may be written to by the debug bus only. When it is written, a reconfiguration is requested. Refer to Sections 6.1 and 6.2 for more information.

4.1.3 CR_CC - Current configuration register



This register is hardwired to the current configuration vector. Refer to Section 6.1 for more information.

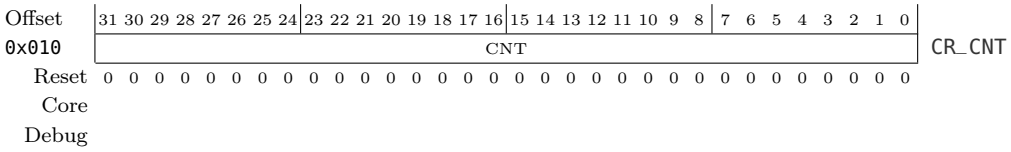
4.1.4 CR_AFF - Cache affinity register



This register stores the cache block index (akin to a lane group) that most recently serviced an instruction fetch for a given context. This may be used for achieving the maximum possible instruction cache locality when reconfiguring.

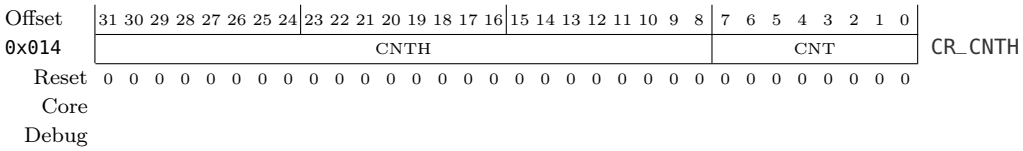
Each nibble represents a lane group. The nibble value is the context index.

4.1.5 CR_CNT - Cycle counter register



Cycle counter. This register is simply always incremented by one in hardware. Simply overflows when it reaches 0xFFFFFFFF. Its intended use is to monitor real time. As an indication, this register overflows approximately every 85 seconds at 50 MHz.

4.1.6 CR_CNTH - Cycle counter register high



This register extends the CR_CNT register by 24 bits. The low byte is equal to the high byte of CR_CNT, similar to the performance counters, which allows the same algorithm to be used in order to read the value. Refer to Section 4.3 for more information. Note however, that unlike the other performance counters, this register always exists, regardless of the design-time configured performance counter width.

4.1.7 CR_LIMCn - Long immediate capability register n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0A0	BORROW15		BORROW14		CR_LIMC7
0x0A4	BORROW13		BORROW12		CR_LIMC6
0x0A8	BORROW11		BORROW10		CR_LIMC5
0x0AC	BORROW9		BORROW8		CR_LIMC4
0x0B0	BORROW7		BORROW6		CR_LIMC3
0x0B4	BORROW5		BORROW4		CR_LIMC2
0x0B8	BORROW3		BORROW2		CR_LIMC1
0x0BC	BORROW1		BORROW0		CR_LIMC0
Reset	* * * * * * * *	* * * * * * * *	* * * * * * * *	* * * * * * * *	
Core					
Debug					

This group of hardwired values represent the supported LIMMH forwarding routes.

BORROW$2n+1$ field, bits 31..16, a.k.a. CR_BORROWi

BORROW$2n$ field, bits 15..0, a.k.a. CR_BORROWi

Each bit in these fields represents a possible LIMMH forwarding route. The bit index within the field specifies the source syllable index, i.e. the LIMMH syllable; $i = (2n, 2n+1)$ is the index of the syllable that uses the immediate.

As an example, if bit 2 in BORROW4 (CR_LIMC2) is set, it means that the third syllable in a bundle (index 2) can be a LIMMH instruction that forwards to the fifth syllable in a bundle (index 4).

For the purpose of generic binaries, the configuration is repeated beyond the number of physically available lanes.

4.1.8 CR_SICn - Syllable index capability register n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0C0	SYL15CAP	SYL14CAP	SYL13CAP	SYL12CAP	CR_SIC3
0x0C4	SYL11CAP	SYL10CAP	SYL9CAP	SYL8CAP	CR_SIC2
0x0C8	SYL7CAP	SYL6CAP	SYL5CAP	SYL4CAP	CR_SIC1
0x0CC	SYL3CAP	SYL2CAP	SYL1CAP	SYL0CAP	CR_SIC0
Reset	0 0 0 0 * * * 1	0 0 0 0 * * * 1	0 0 0 0 * * * 1	0 0 0 0 * * * 1	
Core					
Debug					

This group of hardwired values represent the capabilities of each syllable within a bundle.

SYL$4n+3$CAP field, bits 31..24, a.k.a. CR_SYLiCAP

SYL$4n+2$CAP field, bits 23..16, a.k.a. CR_SYLiCAP

SYL$4n+1$CAP field, bits 15..8, a.k.a. CR_SYLiCAP

SYL$4n$CAP field, bits 7..0, a.k.a. CR_SYLiCAP

Each bit within the field represents a functional unit or resource that is available to syllable index i within a bundle. The following encoding is used.

Bit index	*Function*
0	Always set, indicated that ALU class syllables are supported.
1	If set, multiplier class syllables are supported.
2	If set, memory class syllables are supported.
3	If set, branch class syllables and syllables with stop bits are supported.
4..7	Always zero, reserved for future expansion.

For the purpose of generic binaries, the configuration is repeated beyond the number of physically available lanes.

4.1.9 CR_GPS1 - General purpose register delay register B

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x0D0																																	CR_GPS1
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

This register is reserved for future expansion.

4.1.10 CR_GPS0 - General purpose register delay register A

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x0D4					MEMAR				MEMDC				MEMDR				MULC				MULR				ALUC				ALUR				CR_GPS0
Reset	0	0	0	0	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
Core																																	
Debug																																	

This register lists the key pipeline stages in which the core appears to read from and write to the general purpose register file. Forwarding is taken into consideration, so the core may not actually write to the register file in the listed stages, but from the perspective of the software it seems to.

From these values, the required number of bundles *between* an instruction that writes to a general purpose register and an instruction that reads from one can be determined, being $stage_{commit} - stage_{read} - 1$.

MEMAR field, bits 27..24

Hardwired to the stage in which the memory unit appears to read its address operands from the general purpose registers.

MEMDC field, bits 23..20

Hardwired to the stage in which the memory unit appears to commit the data loaded from memory to the general purpose registers.

MEMDR field, bits 19..16

Hardwired to the stage in which the memory unit appears to read the data to be stored to memory from the general purpose registers.

MULC field, bits 15..12

Hardwired to the stage in which the multiplier appears to commit its result to the general purpose registers.

MULR field, bits 11..8

Hardwired to the stage in which the multiplier appears to read its operands from the general purpose registers.

ALUC field, bits 7..4

Hardwired to the stage in which the ALU appears to commit its result to the general purpose registers.

ALUR field, bits 3..0

Hardwired to the stage in which the ALU appears to read its operands from the general purpose registers.

4.1.11 `CR_SPS1` - Special delay register B

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0D8					CR_SPS1
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core					
Debug					

This register is reserved for future expansion.

4.1.12 `CR_SPS0` - Special delay register A

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x0DC	MEMMC | MEMMR	MEMDC | MEMDR	BRC | BRR	ALUC | ALUR	CR_SPS0
Reset	* * * * * * * *	* * * * * * * *	* * * * * * * *	* * * * * * * *	
Core					
Debug					

This register serves a similar purpose as `CR_GPS0`, but instead of being only for the general purpose registers, these values represents the delay for branch registers, the link register and memory.

MEMMC field, bits 31..28

Hardwired to the stage in which the memory unit actually commits the data from a store instruction to memory.

MEMMR field, bits 27..24

Hardwired to the stage in which the memory unit actually reads the data for a load operation from memory.

MEMDC field, bits 23..20

Hardwired to the stage in which the memory unit appears to commit the data loaded from memory to the link and branch registers.

MEMDR field, bits 19..16

Hardwired to the stage in which the memory unit appears to read the data to be stored to memory from the link and branch registers.

BRC field, bits 15..12

Hardwired to the stage in which the branch unit appears to commit the new program counter. This thus represents the number of branch delay slots. The next instruction is requested in stage 1 and its PC is forwarded combinatorially, thus the number of branch delay slots is $BRC - 2$. Note that the ρ-VEX processor does not actually execute its branch delay slots; it is invalidated when a branch is taken.

BRR field, bits 11..8

Hardwired to the stage in which the branch unit appears to read its operands from the branch and link registers.

ALUC field, bits 7..4

Hardwired to the stage in which the ALU appears to commit its result to the branch and link registers.

ALUR field, bits 3..0

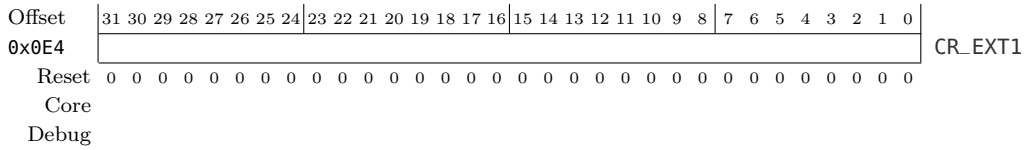
Hardwired to the stage in which the ALU appears to read its operands from the branch and link registers.

4.1.13 CR_EXT2 - Extension register 2

Offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x0E0																																	CR_EXT2
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Core																																	
Debug																																	

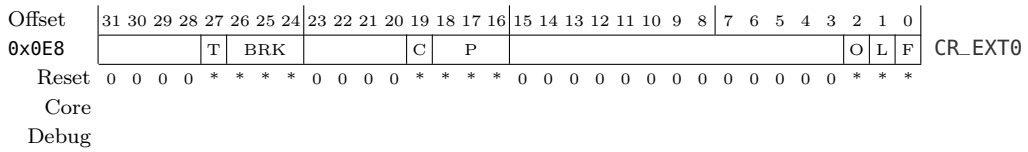
This register is reserved for future expansion.

4.1.14 CR_EXT1 - Extension register 1



This register is reserved for future expansion.

4.1.15 CR_EXT0 - Extension register 0



This register contains flags that specify the supported extensions and quirks of the processor as per its design-time configuration.

T flag, bit 27

Defines whether the trace unit is available. The trace unit has its own capability flags in CR_DCR2.

BRK field, bits 26..24

Defines the number of available hardware breakpoints.

C flag, bit 19

If set, cache-related performance counters exist.

P field, bits 18..16

This field represents the size in bytes of all performance counters except CR_CNT, which is always 64-bit. Refer to Section 4.3 for more information.

O flag, bit 2

This flag determines the unit in which the branch offset field is encoded. When this flag is cleared, the branch offset is encoded in 8-byte units. When it is set, the branch offset is encoded in 4-byte units.

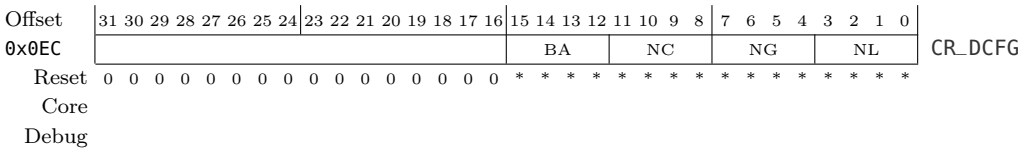
L flag, bit 1

This flag is set when register $r0.63 is mapped to $l0.0, to allow arithmetic to be performed on the link register directly. If it is cleared, these registers are independent.

F flag, bit 0

This flag is set when forwarding is enabled.

4.1.16 CR_DCFG - Design-time configuration register



This register is hardwired to the key parameters that define the size of the processor, such as the number of pipelanes and the number of contexts.

BA field, bits 15..12

Specifies the minimum bundle alignment necessary. Specified as the alignment size in 32-bit words minus 1. For example, if this value is 7, each bundle must start on a 128-byte boundary, as $(7 + 1) \cdot 32 = 128$.

NC field, bits 11..8

Number of hardware contexts supported, minus one.

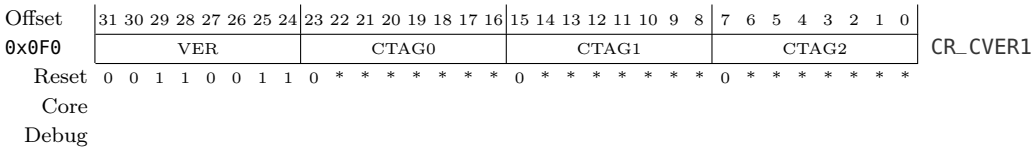
NG field, bits 7..4

Number of pipelane groups supported, minus one. This determines the degree of reconfigurability. Together with NC, it fully specifies the number of valid configuration words.

NL field, bits 3..0

Number of pipelanes in the design, minus one.

4.1.17 CR_CVER1 - Core version register 1



This register specifies the major version of the processor and, together with CR_CVER0, a 7-byte ASCII core version identification tag.

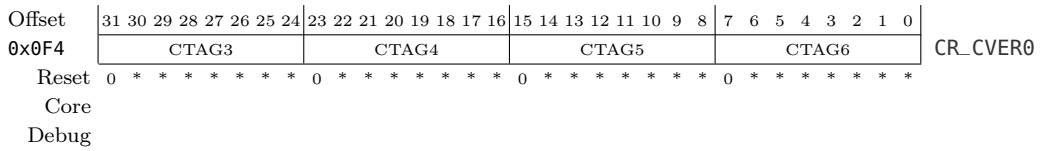
VER field, bits 31..24, a.k.a. CR_CVER

Specifies the major version number of the ρ-VEX processor in ASCII. This will most likely always be '3'.

CTAG0 field, bits 23..16, a.k.a. CR_CTAG

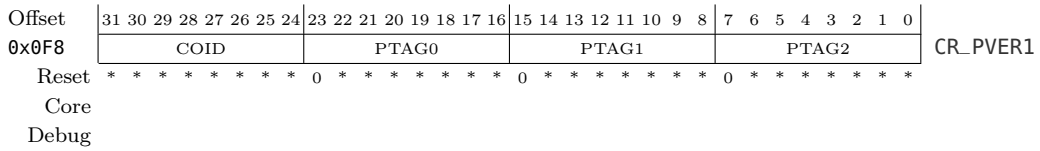
First ASCII character in a string of seven characters, which together identify the core version, similar to how a license plate identifies a car. It is intended that a database will be set up which maps each tag to an immutable archive containing the source code for the core and a mutable errata/notes file.

4.1.18 CR_CVER0 - Core version register 0



Refer to CR_CVER1 for more information.

4.1.19 CR_PVER1 - Platform version register 1



This register specifies the processor index within a platform and, together with CR_PVER0, uniquely identifies the platform using a 7-byte ASCII idenfitication tag.

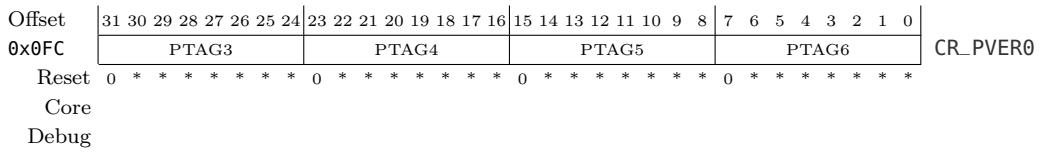
COID field, bits 31..24, a.k.a. CR_COID

Unique processor identifier within a multicore platform.

PTAG0 field, bits 23..16, a.k.a. CR_PTAG

First ASCII character in a string of seven characters, which together identify the platform and bit file, similar to how a license plate identifies a car. It is intended that a database will be set up which maps each tag to an immutable archive containing the source code for the platform, synthesis logs and a bit file, as well as mutable `memory.map`, `rvex.h` and errata/notes files.

4.1.20 CR_PVER0 - Platform version register 0



Refer to CR_PVER1 for more information.

4.2 Context control registers

The following table lists the context control registers of the ρ-VEX processor. The offsets listed are with respect to the control register base address. If you are viewing this manual digitally, you can click the register mnemonics on the right to jump to their documentation.

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x200	CAUSE	BRANCH		K C B R I	CR_CCR
0x204	ID			K C B R I	CR_SCCR
0x208	LR				CR_LR
0x20C	PC				CR_PC
0x210	TH				CR_TH
0x214	PH				CR_PH
0x218	TP				CR_TP
0x21C	TA				CR_TA
0x220	BR0				CR_BR0
0x224	BR1				CR_BR1
0x228	BR2				CR_BR2
0x22C	BR3				CR_BR3
0x230	D J I E R S B	CAUSE	BR3 BR2	BR1 BR0	CR_DCR
0x234	RESULT		TRCAP	T M R C I E	CR_DCR2
...	*Unused*				
0x240	CRR				CR_CRR
0x244	*Unused*				
0x248	WCFG				CR_WCFG
0x24C				RUN S	CR_SAWC
0x250	SCRP1				CR_SCRP1
0x254	SCRP2				CR_SCRP2
0x258	SCRP3				CR_SCRP3
0x25C	SCRP4				CR_SCRP4
0x260	RSC				CR_RSC
0x264	CSC				CR_CSC
0x268	RSC1				CR_RSC1
0x26C	CSC1				CR_CSC1
0x270	RSC2				CR_RSC2
0x274	CSC2				CR_CSC2
0x278	RSC3				CR_RSC3
0x27C	CSC3				CR_CSC3
0x280	RSC4				CR_RSC4
0x284	CSC4				CR_CSC4
0x288	RSC5				CR_RSC5
0x28C	CSC5				CR_CSC5
0x290	RSC6				CR_RSC6
0x294	CSC6				CR_CSC6
0x298	RSC7				CR_RSC7
0x29C	CSC7				CR_CSC7
...	*Unused*				
0x300	CYC3	CYC2	CYC1	CYC0	CR_CYC
0x304	CYC6	CYC5	CYC4	CYC3	CR_CYCH
0x308	STALL3	STALL2	STALL1	STALL0	CR_STALL
0x30C	STALL6	STALL5	STALL4	STALL3	CR_STALLH
0x310	BUN3	BUN2	BUN1	BUN0	CR_BUN
0x314	BUN6	BUN5	BUN4	BUN3	CR_BUNH
0x318	SYL3	SYL2	SYL1	SYL0	CR_SYL

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x31C	SYL6	SYL5	SYL4	SYL3	CR_SYLH
0x320	NOP3	NOP2	NOP1	NOP0	CR_NOP
0x324	NOP6	NOP5	NOP4	NOP3	CR_NOPH
0x328	IACC3	IACC2	IACC1	IACC0	CR_IACC
0x32C	IACC6	IACC5	IACC4	IACC3	CR_IACCH
0x330	IMISS3	IMISS2	IMISS1	IMISS0	CR_IMISS
0x334	IMISS6	IMISS5	IMISS4	IMISS3	CR_IMISSH
0x338	DRACC3	DRACC2	DRACC1	DRACC0	CR_DRACC
0x33C	DRACC6	DRACC5	DRACC4	DRACC3	CR_DRACCH
0x340	DRMISS3	DRMISS2	DRMISS1	DRMISS0	CR_DRMISS
0x344	DRMISS6	DRMISS5	DRMISS4	DRMISS3	CR_DRMISSH
0x348	DWACC3	DWACC2	DWACC1	DWACC0	CR_DWACC
0x34C	DWACC6	DWACC5	DWACC4	DWACC3	CR_DWACCH
0x350	DWMISS3	DWMISS2	DWMISS1	DWMISS0	CR_DWMISS
0x354	DWMISS6	DWMISS5	DWMISS4	DWMISS3	CR_DWMISSH
0x358	DBYPASS3	DBYPASS2	DBYPASS1	DBYPASS0	CR_DBYPASS
0x35C	DBYPASS6	DBYPASS5	DBYPASS4	DBYPASS3	CR_DBYPASSH
0x360	DWBUF3	DWBUF2	DWBUF1	DWBUF0	CR_DWBUF
0x364	DWBUF6	DWBUF5	DWBUF4	DWBUF3	CR_DWBUFH

4.2.1 CR_CCR - Main context control register

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0							
0x200	CAUSE	BRANCH		K	C		B		R		I	CR_CCR
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 1	1	0	1	0	1	0	1	0
Core					✓	✓	✓	✓	✓	✓	✓	✓
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓			✓	✓	✓	✓	✓	✓	✓	✓

The primary purpose of the context control register is to store the primary control flags of the processor, for example whether interrupts are enabled. In addition, it also stores the trap cause and exposes the branch register file to the debug bus.

CAUSE field, bits 31..24, a.k.a. CR_TC

Trap cause. Set to the trap cause by hardware when the trap handler is called. Reset to 0 by hardware when an RFI instruction is encountered. Read-write by the debug bus, but the processor cannot write to this register.

BRANCH field, bits 23..16, a.k.a. CR_BR

Branch register file. Contains the current state of the branch registers. Only intended for use by the debug bus to see and modify the state of the branch register file. While the core is running, accessing this register is undefined due to it being dependent on the pipeline and forwarding state.

K field, bits 9..8

This register selects between kernel mode and user mode. Kernel mode is activated when the core is reset and when entering the trap or panic handlers. These must thus always

point to code in hardware memory space. When RFI is executed, the state is restored from CR_SCCR.

In kernel mode, the register reads as 01, while in user mode, it reads as 10. The only way to enter user mode is by writing the user mode command to CR_SCCR and subsequently executing RFI. Neither the core nor the debug bus can write to this field directly.

Currently, the status of the kernel mode flag has no effect on the ρ-VEX. However, it is intended that this register will be used in the future for memory protection and/or security features. In particular, the reason that this flag can only be set by entering a trap and cleared by executing RFI, is that both of these mechanisms can be configured to do a full pipeline flush, which allows this flag to control whether address translation is enabled or disabled.

C field, bits 7..6

This register controls whether the context switch trap is enabled. It does not exist on hardware context 0. When the core is reset or the trap service routine is entered, the context switch trap is disabled. When RFI is executed, the state is restored from CR_SCCR.

When the context switch trap is enabled, this register reads as 01. When it is disabled, it reads as 10. Both the core and the debug bus can write to this register. Writing 00 has no effect, writing 01 enables the context switching trap, writing 10 disables it and writing 11 toggles the state. This prevents the need for read-modify-write operations.

Refer to CR_RSC for more information.

B field, bits 5..4

This register controls whether breakpoints are enabled in self-hosted debug mode. Its value is ignored in external debug mode. When the core is reset or the trap service routine is entered due to a debug trap in self-hosted debug mode, breakpoints are disabled. When RFI is executed, the state is restored from CR_SCCR.

When breakpoints are enabled, this register reads as 01. When they are disabled, it reads as 10. Both the core and the debug bus can write to this register. Writing 00 has no effect, writing 01 enables debug traps, writing 10 disables them and writing 11 toggles the state. This prevents the need for read-modify-write operations.

R field, bits 3..2

This register, named ready-for-trap, tentatively specifies if the processor is currently capable of servicing traps. However, since traps cannot be masked, any trap that occurs while ready-for-trap is cleared will cause a panic. Therefore, the only thing this register does in hardware is switch between the trap handler and panic handler address. When the core is reset or the trap service routine is entered, ready-for-trap is cleared. When RFI is executed, the state is restored from CR_SCCR.

When ready-for-trap is set (trap handler selected), this register reads as 01. When it is cleared (panic handler selected), it reads as 10. Both the core and the debug bus can write to this register. Writing 00 has no effect, writing 01 sets ready-for-trap, writing 10

clears it and writing 11 toggles the state. This prevents the need for read-modify-write operations.

I field, bits 1..0

This register selects whether external interrupts are enabled or not. When the core is reset or the trap service routine is entered, external interrupts are disabled. When RFI is executed, the state is restored from CR_SCCR.

When interrupts are enabled, this register reads as 01. When they are disabled, it reads as 10. Both the core and the debug bus can write to this register. Writing 00 has no effect, writing 01 enables external interrupts, writing 10 disables them and writing 11 toggles the state. This prevents the need for read-modify-write operations.

4.2.2 CR_SCCR - Saved context control register

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10	9 8	7 6	5 4	3 2	1 0	
0x204	ID			K	C	B	R	I	CR_SCCR
Reset	* * * * * * * *	0 0 0 0 0 0 0 0	0 0 0 0 0 0	1 0	1 0	1 0	1 0	1 0	
Core				✓ ✓	✓ ✓	✓ ✓	✓ ✓	✓ ✓	
Debug				✓ ✓	✓ ✓	✓ ✓	✓ ✓	✓ ✓	

This register saves the state of the primary control flags of the processor when entering the trap service routine. When RFI is executed, the state is restored from this register. In addition, this register contains the context ID, which contexts may read to identify themselves.

ID field, bits 31..24, a.k.a. CR_CID

This field is hardwired to the context index. Programs running on the ρ-VEX processor may use this field to determine which hardware context they are running on.

Note that CR_CID is not unique in a multi-processor system. If a unique processor ID is needed in such a case, CR_COID should be used as well.

K field, bits 9..8

When the trap service routine is entered, this register stores whether kernel the processor was in kernel mode or user mode. When RFI is executed, the state is set to this value.

Unlike the kernal mode field in CR_CCR, this field can be written. Writing 00 has no effect, writing 01 selects kernel mode, writing 10 selects user mode and writing 11 toggles the state. This prevents the need for read-modify-write operations. Read behavior is identical to the K field in CR_CCR.

C field, bits 7..6

When the trap service routine is entered, this register stores whether the context switching trap was enabled. When RFI is executed, the state is set to this value.

Core and debug bus access behavior is identical to the C field in CR_CCR.

B field, bits 5..4

When the trap service routine is entered, this register stores whether self-hosted debug breakpoints were enabled. When RFI is executed, the state is set to this value.

Core and debug bus access behavior is identical to the B field in CR_CCR.

R field, bits 3..2

When the trap service routine is entered, this register stores whether ready-for-trap was set. When RFI is executed, the state is set to this value.

Core and debug bus access behavior is identical to the R field in CR_CCR.

I field, bits 1..0

When the trap service routine is entered, this register stores whether interrupts were enabled. When RFI is executed, the state is set to this value.

Core and debug bus access behavior is identical to the I field in CR_CCR.

4.2.3 CR_LR - Link register

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x208	LR				CR_LR
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core					
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

Contains the current link register ($l0.0) value. Only intended for use by the debug bus. While the core is running, accessing this register is undefined due to it being dependent on the pipeline and forwarding state.

4.2.4 CR_PC - Program counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x20C	PC				CR_PC
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core					
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

Contains the current program counter. Only intended for use by the debug bus. When the register is written by the debug bus, the jump flag in CR_DCR is set, to ensure that the branch unit properly jumps to the new PC. This works even if the processor is running.

4.2.5 CR_TH - Trap handler

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x210	TH				CR_TH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

Contains the address of the trap service routine. This is where the processor will jump to if a trap occurs while ready-for-trap in CR_CCR is set. Even if the design contains an MMU, this should be a hardware address, as the MMU is disabled when a trap occurs.

4.2.6 CR_PH - Panic handler

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x214	PH				CR_PH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

Contains the address of the panic service routine. This is where the processor will jump to if a trap occurs while ready-for-trap in CR_CCR is NOT set. Even if the design contains an MMU, this should be a hardware address, as the MMU is disabled when a trap occurs.

The difference between the trap and panic service routines, is that the trap service routine has all state information of the processor at its disposal. That is, if the trap is recoverable, the program can continue after the trap service routine completes. The panic service routine, however, should assume that the state information of the processor is incomplete. Refer to Section 5 for more information.

4.2.7 CR_TP - Trap point

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x218	TP				CR_TP
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

When a trap occurs, this register is set to the address of the start of the offending bundle. The address is in user space if the MMU was enabled when the trap occured. In addition, when RFI is executed, the processor will jump back to this address to resume execution. This is the correct behavior for both external interrupts and traps that, after servicing, should return to the previously offending instruction, such as a page fault.

To support software context switching, the processor may write to this register to change the resumption address. RFI will then cause execution to be resumed in the new software context, assuming the rest of the processor state has been swapped in as well.

4.2.8 CR_TA - Trap argument

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x21C	TA				CR_TA
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core					
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

When a trap occurs, this register is set to the trap argument. The significance of this value depends on the trap, which can be identified from the trap cause field in CR_CCR. Refer to Section 5 for more information.

4.2.9 CR_BRn - Breakpoint n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x220	BR0				CR_BR0
0x224	BR1				CR_BR1
0x228	BR2				CR_BR2
0x22C	BR3				CR_BR3
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

These registers hold the addresses for the hardware breakpoints and/or watchpoints. These registers only exist up to how many break-/watchpoints are design-time configured to be supported by the processor. The functionality of the breakpoints is configured in CR_DCR. These registers are always writable by the debug bus, but they are only writable by the core when the E flag is cleared, i.e. when self-hosted debug mode is selected.

4.2.10 CR_DCR - Debug control register 1

Offset	31	30	29	28	27	26	25	24	23 22 21 20 19 18 17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0	
0x230	D	J		I	E	R	S	B	CAUSE		BR3		BR2		BR1		BR0	CR_DCR
Reset	0	0	0	1	0	0	0	0	0 0 0 0 0 0 0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	
Core							✓				✓ ✓		✓ ✓		✓ ✓		✓ ✓	
Debug	✓			✓	✓	✓	✓	✓			✓ ✓		✓ ✓		✓ ✓		✓ ✓	

This register controls the debugging system of the ρ-VEX processor.

D flag, bit 31

Done/reset flag. This bit is set by hardware when a STOP instruction is encountered. It is cleared when a one is written to the R or S flags.

In addition, when a one is written to this flag, the control register file for this context is completely reset, as if the external context reset signal was asserted. Writing a zero has no effect. When combined with writing a one to the external debug flag, the core starts in external debug mode, and when combined with writing a one to B or the S flag, the core will stop execution before any instruction is executed, allowing the user to single-step from the start of the program. This works because I, E, S and B are not affected by a context reset.

Note that breakpoint information will have to be reloaded when the context is reset using this method.

J flag, bit 30

This bit is set by hardware when the debug bus writes to the PC register and is cleared when the processor jumps to it. It can thus be used as an acknowledgement flag for jumping. The flag is read only.

I flag, bit 28

Internal debug flag. Complement of the external debug flag. When the debug bus writes a one to this flag, the external debug flag is cleared, giving the processor control over

debugging. Writing a zero has no effect. This flag is not affected by a context reset; it is only reset when the entire core is reset.

E flag, bit 27

External debug flag. Complement of the internal debug flag. When the debug bus writes a one to this flag, the external debug flag is set, enabling external debug mode. Writing a zero has no effect. This flag is not affected by a context reset; it is only reset when the entire core is reset.

While in external debug mode, debug traps cause the B flag to be set and the trap cause to be recorded in `CR_DCR` instead of the normal registers. This thus allows an external debugger to handle the debug traps instead, even if the processor is in the middle of a trap service routine and is not even ready for a trap. Writing a one to the R or the S flag is the equivalent of `RFI` for the external debugging system.

R flag, bit 26

Resume flag. When the debug bus writes a one to this flag, the B flag is cleared, causing the processor to resume execution if it was halted. Writing a zero has no effect; this flag is cleared by hardware when the first instruction is successfully fetched. It can thus be used as an acknowledgement flag for resuming execution.

In addition, debug traps are disabled for instructions that were fetched while this flag was set. This behavior allows the processor to step beyond the breakpoint that caused the processor to break, so there is no need to disable the triggered breakpoint in order to resume. This behavior is also used for single stepping; see below.

S flag, bit 25

Step flag. This flag may be set by the debug bus by writing a one to it. Doing so will also cause the R flag to be set and the B flag to be cleared, causing the processor to resume execution if it was halted. Writing a zero has no effect. The processor can also set this flag, but only if the E flag is cleared, i.e., if the processor is in self-hosted debug mode. This flag is not affected by a context reset; it is only reset when the entire core is reset.

While set, any instruction will cause a step debug trap. However, as noted above, all debug traps are disabled for the first instruction fetched after execution resumes. They should also be disabled while in the trap service routine through the breakpoint enable field in `CR_CCR`. This allows both an external debugger and the self-hosted debug system to single-step.

B flag, bit 24

Break flag. When this flag is set, the context stops fetching instructions and flushes the pipeline, as it would if the external run signal is low or if a reconfiguration is pending. It effectively halts execution. This flag is not affected by a context reset; it is only reset when the entire core is reset.

This flag may be set by the debug bus by writing a one to it, in order to pause execution. Writing a zero has no effect. In addition, the flag is set by hardware when a debug trap occurs while the E flag is set and when a `STOP` instruction is executed.

CAUSE field, bits 23..16, a.k.a. `CR_DCRC`

Trap cause for debug traps that should be handled by the external debug system. This is set to the debug trap cause by hardware when the B flag is set due to a debug trap. It is cleared when a one is written to resume or step, and set to `0x01` if a one is written to the B flag.

BR3 field, bits 13..12

Breakpoint 3 control field. This field only exists if the core is design-time configured to support all four hardware breakpoints. See also BR0.

BR2 field, bits 9..8

Breakpoint 2 control field. This field only exists if the core is design-time configured to support at least three hardware breakpoints. See also BR0.

BR1 field, bits 5..4

Breakpoint 1 control field. This field only exists if the core is design-time configured to support at least two hardware breakpoints. See also BR0.

BR0 field, bits 1..0

Breakpoint 0 control field. This field only exists if the core is design-time configured to support at least one hardware breakpoint.

The core can only write to BRn fields when the E flag is cleared, i.e. when self-hosted debug mode is selected. The encoding for the fields is as follows.

BRn = `00`: breakpoint/watchpoint disabled.
BRn = `01`: breakpoint enabled.
BRn = `10`: data write watchpoint enabled.
BRn = `11`: data read/write watchpoint enabled.

4.2.11 `CR_DCR2` - Debug control register 2

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x234	RESULT		TRCAP	T M R C I E	CR_DCR2
Reset	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0	* * * * * 0 0 *	0 0 0 0 0 0 0 0	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓			✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓			✓ ✓ ✓ ✓ ✓ ✓	

This register controls the trace unit, if the core is design-time configured to support tracing. It also contains an 8-bit scratchpad register for communicating an execution result to the debug system.

RESULT field, bits 31..24, a.k.a. CR_RET

This field does not have a hardwired function. It is intended to be used to communicate the reason for executing a STOP instruction to the debug system. The default _start.s file will write the main() return value to this register before stopping.

TRCAP field, bits 15..8

This field lists the tracing capabilities of the core. The bit indices in this byte correspond to the bit indices in the control byte (the least significant byte of CR_DCR2). If a bit is high, the feature is available.

T flag, bit 7

Setting this bit enables trap tracing if the E flag is set and the core is design-time configured to support it.

M flag, bit 6

Setting this bit enables memory/control register tracing if the E flag is set and the core is design-time configured to support it.

R flag, bit 5

Setting this bit enables register write tracing if the E flag is set and the core is design-time configured to support it.

C flag, bit 4

Setting this bit enables cache performance tracing if the E flag is set and the core is design-time configured to support it.

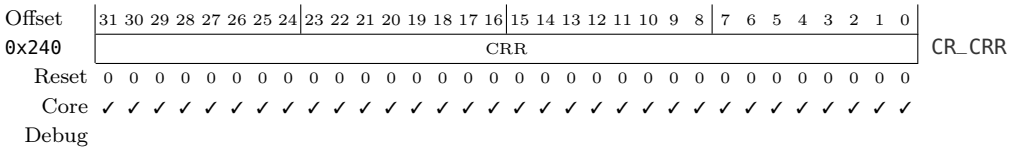
I flag, bit 3

Setting this bit causes all fetched instructions to be traced if the E flag is set and the core is design-time configured to support it.

E flag, bit 0

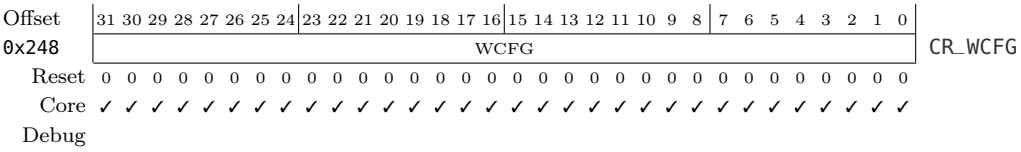
Setting this bit enables tracing if the core is design-time configured to support it. If no other bits are set, only branch origins and destinations are traced.

4.2.12 CR_CRR - Context reconfiguration request register



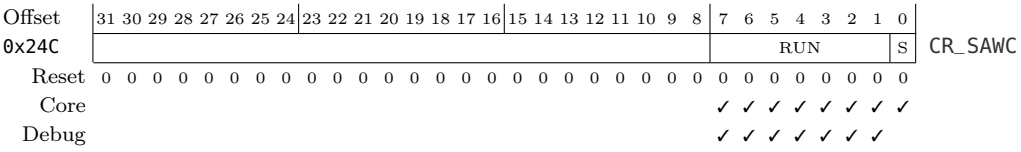
This register may be written to by the core only. When it is written, a reconfiguration is requested. Refer to Section 6 for more information.

4.2.13 CR_WCFG - Wakeup configuration



This register only exists on context 0. This configuration register is used in conjunction with the S flag in CR_SAWC. Refer to Section 6.3 for more information.

4.2.14 CR_SAWC - Sleep and wake-up control register



This register only exists on context 0. This register contains special control features for sleeping (reconfiguring to a configuration with all lane groups disabled) and waking up other hardware contexts.

RUN field, bits 7..1

This field contains a bit for every other context, i.e., not all of these bits will be available if the core is not configured to support all eight hardware contexts. When reading this register, each bit represents the ones complement of the B flag in CR_DCR for each other context. Writing a one to a bit is equivalent to writing a one to the R flag in CR_DCR for each other context.

A scheduler running on context 0 may use this feature, combined with an interrupt controller that triggers an interrupt when the done output for any other context has a rising edge, to support task yielding for cooperative scheduling. A yield will then be equivalent to a STOP instruction, which will thus trigger an interrupt for the scheduler. The scheduler may then switch out the software context and subsequently restart the hardware context using these flags.

S flag, bit 0

Sleep flag. This enables or disables the sleep and wake-up system. Refer to Section 6.3 for more information.

4.2.15 CR_SCRPn - Scratchpad register n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x250	SCRP1				CR_SCRP1
0x254	SCRP2				CR_SCRP2
0x258	SCRP3				CR_SCRP3
0x25C	SCRP4				CR_SCRP4
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

Scratch pad registers. May be used at the discretion of the application and/or debug system.

4.2.16 CR_RSC - Requested software context

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x260	RSC				CR_RSC
Reset	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This register does not exist on context 0. It is hardwired to RSCn in hardware context 0, and represents the software context that should be loaded into our hardware context, if it is not already loaded. The encoding of the register is at the user's discretion, but it is intended that this points to a memory region that contains the to be loaded context.

The contents of this register may also be written by hardware context 0 through RSCn, which is expected to run the scheduler. When this value does not equal the value in CSC and context switching is enabled in CR_CCR, the TRAP_SOFT_CTXT_SWITCH trap is caused. Refer to its documentation in Section 5 for more information.

4.2.17 CR_CSC - Current software context

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x264	CSC				CR_CSC
Reset	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This register does not exist on context 0. It is hardwired to CSCn in hardware context 0. The value in this register should be set to the value in CR_RSC by the TRAP_SOFT_CTXT_SWITCH trap.

4.2.18 CR_RSCn - Requested swctxt on hwctxt n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x268	RSC1				CR_RSC1
0x270	RSC2				CR_RSC2
0x278	RSC3				CR_RSC3
0x280	RSC4				CR_RSC4
0x288	RSC5				CR_RSC5
0x290	RSC6				CR_RSC6
0x298	RSC7				CR_RSC7
Reset	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	
Core	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This register only exists on context 0, and only if the core is design-time configured to support hardware context n. This register is hardwired to CR_RSC in hardware context n. Refer to CR_RSC for more information.

4.2.19 CR_CSCn - Current swctxt on hwctxt n

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x26C	CSC1				CR_CSC1
0x274	CSC2				CR_CSC2
0x27C	CSC3				CR_CSC3
0x284	CSC4				CR_CSC4
0x28C	CSC5				CR_CSC5
0x294	CSC6				CR_CSC6
0x29C	CSC7				CR_CSC7
Reset	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1	
Core					
Debug					

This register only exists on context 0, and only if the core is design-time configured to support hardware context n. This register is hardwired to CR_CSC in hardware context n. Refer to CR_CSC for more information.

4.2.20 CR_CYC - Cycle counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x300	CYC3	CYC2	CYC1	CYC0	CR_CYC
0x304	CYC6	CYC5	CYC4	CYC3	CR_CYCH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every cycle while an instruction from this context is in the pipeline, even when the context is stalled.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.21 CR_STALL - Stall cycle counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x308	STALL3	STALL2	STALL1	STALL0	CR_STALL
0x30C	STALL6	STALL5	STALL4	STALL3	CR_STALLH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every cycle while an instruction from this context is in the pipeline and the context is stalled. As long as neither CR_CYC nor CR_STALL have overflowed, CR_CYC - CR_STALL represents the number of active cycles.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.22 CR_BUN - Committed bundle counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x310	BUN3	BUN2	BUN1	BUN0	CR_BUN
0x314	BUN6	BUN5	BUN4	BUN3	CR_BUNH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments whenever the results of executing a bundle are committed. As long as neither CR_CYC, CR_STALL nor CR_BUN have overflowed, CR_CYC - CR_STALL - CR_BUN represents the number of cycles spent doing pipeline flushes, for example due to traps or the branch delay slot.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.23 CR_SYL - Committed syllable counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x318	SYL3	SYL2	SYL1	SYL0	CR_SYL
0x31C	SYL6	SYL5	SYL4	SYL3	CR_SYLH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments whenever the results of executing a non-NOP syllable are committed. As long as neither CR_BUN nor CR_SYL have overflowed, CR_SYL / CR_BUN represents average instruction-level parallelism since the registers were cleared.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.24 CR_NOP - Committed NOP counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x320	NOP3	NOP2	NOP1	NOP0	CR_NOP
0x324	NOP6	NOP5	NOP4	NOP3	CR_NOPH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments whenever a NOP syllable is committed. As long as neither CR_SYL nor CR_NOP have overflowed, CR_SYL / (CR_SYL + CR_NOP) represents average

fraction of syllables that are `NOP`, i.e. the compression efficiency of the binary.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.25 `CR_IACC` - Instruction cache access counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x328	IACC3	IACC2	IACC1	IACC0	CR_IACC
0x32C	IACC6	IACC5	IACC4	IACC3	CR_IACCH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments for every instruction cache access.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.26 `CR_IMISS` - Instruction cache miss counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x330	IMISS3	IMISS2	IMISS1	IMISS0	CR_IMISS
0x334	IMISS6	IMISS5	IMISS4	IMISS3	CR_IMISSH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a miss in the instruction cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.27 `CR_DRACC` - Data cache read access counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x338	DRACC3	DRACC2	DRACC1	DRACC0	CR_DRACC
0x33C	DRACC6	DRACC5	DRACC4	DRACC3	CR_DRACCH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a read access to the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.28 `CR_DRMISS` - Data cache read miss counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x340	DRMISS3	DRMISS2	DRMISS1	DRMISS0	CR_DRMISS
0x344	DRMISS6	DRMISS5	DRMISS4	DRMISS3	CR_DRMISSH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a read miss in the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.29 CR_DWACC - Data cache write access counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x348	DWACC3	DWACC2	DWACC1	DWACC0	CR_DWACC
0x34C	DWACC6	DWACC5	DWACC4	DWACC3	CR_DWACCH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a write access to the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.30 CR_DWMISS - Data cache write miss counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x350	DWMISS3	DWMISS2	DWMISS1	DWMISS0	CR_DWMISS
0x354	DWMISS6	DWMISS5	DWMISS4	DWMISS3	CR_DWMISSH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a write miss in the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.31 CR_DBYPASS - Data cache bypass counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x358	DBYPASS3	DBYPASS2	DBYPASS1	DBYPASS0	CR_DBYPASS
0x35C	DBYPASS6	DBYPASS5	DBYPASS4	DBYPASS3	CR_DBYPASSH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time there is a bypassed access to the data cache.

Refer to Section 4.3 for more information about the structure of performance counters.

4.2.32 CR_DWBUF - Data cache write buffer counter

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
0x360	DWBUF3	DWBUF2	DWBUF1	DWBUF0	CR_DWBUF
0x364	DWBUF6	DWBUF5	DWBUF4	DWBUF3	CR_DWBUFH
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	
Core				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	
Debug				✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓	

This performance counter increments every time the cache has to wait for the write buffer to flush in order to process the current request.

Refer to Section 4.3 for more information about the structure of performance counters.

4.3 Performance counter registers

All performance counters share the same nontrivial 64-bit structure, representing up to 56 bits worth of counter data. The actual size is design-time configurable using the CFG

vector, and may be read from field P in `CR_EXT0`.

Each performance counter may be reset independently by writing an even value to the low register. Alternatively, all context-specific performance counters may be reset at the same time by writing an odd number to one of the performance counter low registers.

64-bit reads cannot be performed atomically in the ρ-VEX. Therefore, reliably reading the performance counters when they are configured to be larger than a 32-bit word is impossible to do in general, without additional hardware.

Typically, a holding register is implemented for either the low or the high word, which is loaded at the exact same time the other word is read. While this is fine in a single-processor environment, a multiprocessor environment would need such a holding register for each processor separately. To make matters worse, this holding register would also need to be saved and restored when a software context is swapped out. This makes this solution more trouble than it's worth.

In the ρ-VEX, this problem is not avoided completely, but it is mitigated. Each counter is limited to seven bytes, and the middle byte is mirrored by both the low and high register if the counter is larger than a 32-bit word. This permits the following algorithm for a semi-reliable performance counter read.

```
/**
 * Loads a 40-bit, 48-bit or 56-bit performance counter value. Do not use this
 * when the counter size is set to 32-bit!
 */
uint64_t read_counter(
    volatile uint32_t *low,
    volatile uint32_t *high
) {

    // Perform the read.
    uint32_t l = *low;
    uint32_t h = *high;

    // Check if the counters have overflowed.
    if (l >> 24) != (h & 0xFF) {

        // There was an overflow, so clear the low value.
        l = 0;

    }

    // Combine the values and return.
    return ((uint64_t)h << 24) | l;

}
```

Note that this algorithm will *not* work when the counters are configured to be 32 bits wide. In this case the high word register is intentionally not implemented in order to save hardware, which means that the overflow check will not work properly.

The algorithm assumes that the value is monotonously increasing. This is true for all performance counters as long as it is impossible for them to be reset during the read. As long as there was no 32-bit overflow during the read, the returned value will always be a counter value between what is was when `low` was read and what it was when `high` was

read. If there *was* such an overflow, there is a small chance (1/256 if the added value during the read would be uniformly distributed) that the returned value is slightly higher than what the counter value was when `high` was read.

As an example, the worst case scenario is that the counter is at 0xFFFFFF when `low` is read (`l` = 0xFFFFFF), and at 0x100000000 when `high` is read (`h` = (`l` = 0x100). This will result in 0x100FFFFFF being returned, or about 0.4% too much. This is, however, completely insignificant compared to the jitter which may be expected in the value when such a delay is possible between the two reads. It would require an extremely long interrupt service routine or software context switch happening at exactly the wrong time, and when such things are going on in the background.

5 Traps and interrupts

There are many systems in a processor that need to be able to interrupt normal program flow. For instance, an external interrupt may be requested, or a problem occured while trying to load a word from memory, such as a page fault. The naming conventions for such interruptions varies from processor to processor; in the ρ-VEX processor all such interruptions are called traps. The word 'interrupt' is reserved for the special trap that deals with external interrupts, i.e. asynchronous signals from outside the core. The word 'fault' is used to refer to traps that signal that an instruction could not be executed.

5.1 Trap sources

There are roughly six sources of traps in the ρ-VEX processor, which are handled in slightly different ways.

- Faults. A fault signals that an instruction could not be executed for some reason. They are always handled by the processor by jumping to the trap or panic handler. With the exception of page faults, these traps are usually non-recoverable, leading to abnormal termination of the executing task in an operating system environment, or the `STOP` instruction to be called in a bare-metal environment.
- Interrupts. The ρ-VEX processor core has an interface for an interrupt controller. When an interrupt is requested and interrupts are enabled through the I flag in `CR_CCR`, a `TRAP_EXT_INTERRUPT` trap will be generated. This trap causes the processor to jump to the trap or panic handler.
- Context switch request. When the values in `CR_RSC` and `CR_CSC` do not match and the context switching system is enabled by means of the the C flag in `CR_CCR`, a `TRAP_SOFT_CTXT_SWITCH` trap will be generated. This trap causes the processor to jump to the trap or panic handler.
- Breakpoints/debug traps. When a hardware or software breakpoint is hit while breakpoints are enabled through the B flag in `CR_CCR`, the processor will generate a debug trap. Debug traps can be handled in two ways, depending on the E and I flags in `CR_DCR`. When the I flag is set (this is the default), the traps will be handled as any other trap, i.e. by jumping to the trap or panic handler. However, when the E flag is set, the context will simply halt, and write the trap cause to the cause field in `CR_DCR`. This allows an external debugging system to handle the breakpoints instead of the processor itself. In addition, debug traps are disabled for every first instruction executed after returning from a trap handler or restarting the context, allowing either to jump over breakpoints.

- TRAP instructions. The TRAP instruction can be used to emulate any trap. If the cause maps to a debug trap, it is handled exactly as a debug trap, allowing it to be used as a software breakpoint. Otherwise, it is handled like a fault.

- STOP instructions. A STOP instruction halts the core by generating a TRAP_STOP trap during execution of the subsequent instruction. The TRAP_STOP trap is always handled by stopping the hardware context. In addition, the D flag in CR_DCR is set and the done output signal for the stopped hardware context is asserted.

5.2 Trap and panic handlers

As stated above, most traps are handled by jumping to the so-called trap or panic handler. These handlers are simply subroutines that typically end with either a RFI or STOP instruction. They should be pointed to by the CR_TH and CR_PH control registers; it is up to the initialization code to set up these links.

The hardware switches between the trap and panic handlers based on the R flag in CR_CCR. The only hardware difference between them is that this flag always switches to the panic handler upon servicing a trap, such that a trap that immediately follows another trap will always be handled by the panic handler.

The tentative difference between the two trap handlers is that one should attempt to jump back to the application (or alternatively, in the case of an operating system, kill the current process with the appropriate signal and context switch to another thread), and the other should not. The necessity of such a difference can be best illustrated with a simple example.

Consider a program that has just been trapped due to an interrupt. The first course of action in handling a trap must always be to save the state of the running program, so the trap cause and argument registers can be examined. Now consider that it is possible for these context-saving memory accesses to cause, say, a misaligned memory access, due to a programming error. A regular trap handler may in theory try to recover from the fault by emulating the faulting instruction and then jumping over it. However, if it would do so, the trap point, cause and argument control registers of the original interrupt trap will have been overwritten with the misaligned memory access. This data was simply lost when the second trap occured; there is no way around this. Thus, the program cannot continue.

Through the dual handler system as implemented in the ρ-VEX processor, the first trap will be handled by the regular trap handler. Upon jumping to this trap handler, the processor will automatically clear the ready-for-trap flag, such that the second trap will be handled by the panic handler.

What it comes down to, is that the trap handler may try to recover from a fault, handle an interrupt or breakpoint, etc., while the panic handler should simply display or log an error message if it can, and then stop execution or reset. The regular trap handler may also want to jump to the panic handler if it is posed with a fault trap that it cannot recover from.

5.3 Trap identification

In the ρ-VEX processor, traps are identified not by the address that the processor branches to (as there are only two of these addresses, as described in the previous section), but by the trap cause (CR_TC) and trap argument (CR_TA) control registers. The former stores an 8-bit value that identifies the cause of the trap. The latter is a 32-bit register whose significance depends on the trap cause.

The list below documents the trap causes as currently defined in the processor, and the significance of the trap argument. Note, however, that a TRAP instruction may emulate any of these traps with any argument.

- TRAP_NONE = 0x00

  Trap cause 0 is reserved to indicate normal operation. When an RFI instruction is executed, the trap cause register (cause field in CR_CCR) will be reset to 0, so an external debug system can always determine what a program is doing, unless nested traps are utilized.

- TRAP_INVALID_OP = 0x01

  This trap is generated by hardware in the following conditions.

  - An unknown opcode is encountered.
  - The stop bit was set such that the next bundle would start on an address violating the minimum design-time configured bundle alignment.
  - A branch opcode is encountered in a pipelane that does not have an active branch unit.
  - A memory opcode is encountered in a pipelane that is not design-time configured to include a memory unit.
  - A multiplier opcode is encountered in a pipelane that is not design-time configured to include a multiplier.

  The trap argument is set to the lane index that caused the trap.

- TRAP_MISALIGNED_BRANCH = 0x02

  This trap is generated by hardware when a branch to a misaligned address is requested. The trap argument is set to the branch target.

- TRAP_FETCH_FAULT = 0x03

  This trap is generated by hardware when an instruction fetch resulted in a bus fault. The trap argument is unused; the program counter can be determined from the trap point.

- TRAP_MISALIGNED_ACCESS = 0x04

  This trap is generated by hardware when a misaligned memory access was requested. That is, a 32-bit word access was attempted with an address that is not

divisible by four, or a 16-bit word access was attempted with an odd address. The trap argument is set to the requested memory address.

- TRAP_DMEM_FAULT = 0x05

  This trap is generated by hardware when a data memory access resulted in a bus fault. The trap argument is set to the requested memory address.

- TRAP_LIMMH_FAULT = 0x06

  This trap is generated by hardware under the following conditions.

  - A LIMMH instruction is trying to forward to a lane for which no route is available in the core. Note that only the least significant bit of the target lane is actually checked, though. In this case, the trap argument is the index of the lane with the LIMMH instruction.
  - Two LIMMH instructions are trying to forward to the same lane. In this case, the trap argument is the index of the target lane.
  - A LIMMH instruction is attempting to forward to a syllable that is not using an immediate. In this case, the trap argument is also the index of the target lane.

- TRAP_EXT_INTERRUPT = 0x07

  This trap is generated by hardware when the external interrupt request line is asserted while interrupts are enabled by means of the I flag in CR_CCR. When the trap service routine is entered, the state of the external interrupt ID signal is saved as the trap argument in CR_TA, and in the same cycle, the interrupt is acknowledged. This ensures that the interrupt ID presented to the trap service routine always matches the acknowledged interrupt.

  There is a delay between the core registering that the external interrupt request line is asserted and generating the trap, and the actual entering of the trap service routine. This delay is due to the pipeline flush required to do this, and is in the order of a couple cycles; compared to actually servicing a trap this delay is negligible. However, if it is ever possible that an active interrupt is disabled before it is acknowledged by the core, it is possible that the core will enter the trap service routine due to an interrupt that was disabled before it could be handled. In this case, the interrupt controller should provide the core with an otherwise reserved interrupt ID indicating that there was no interrupt. The trap service routine should handle this special interrupt ID as no-operation.

- TRAP_STOP = 0x08

  This trap is generated by hardware in the instruction immediately following a STOP instruction. It is handled in a completely different way than the other traps are; the hardware will not jump to CR_TH or CR_PH. Instead, the D and B flags in CR_DCR are set, thus stopping execution, and the program counter is set to the trap point. This allows an external debugging or control system to resume processing after the stop trap by simply writing a one to the R flag in CR_DCR.

- TRAP_SOFT_CTXT_SWITCH = 0x09

  This trap is generated by hardware when the contents of CR_RSC differ from CR_CSC while this trap is enabled using the C flag in CR_CCR. The intended use of this trap is to allow hardware context 0 to control software context switching on the other hardware contexts. When used in this way, the trap service routine for this trap should perform the following tasks.

  - If CR_CSC $\neq$ -1, save the current context to the memory identified by CR_CSC.
  - Set CR_CSC to CR_RSC.
  - Restore the software context identified by CR_RSC from memory.

  The way in which CR_RSC and CR_CSC identify the software context to be exchanged is up to the operating system code.

- TRAP_SOFT_DEBUG_0 = 0xF8
- TRAP_SOFT_DEBUG_1 = 0xF9
- TRAP_SOFT_DEBUG_2 = 0xFA

  These traps are never generated by hardware, but are intended to be used as soft breakpoints using the TRAP instruction. That is, the debug system may override one of the syllables in a any bundle where a breakpoint is desired with a TRAP syllable. It may return control to the application by reverting the TRAP syllable back into the original syllable. If it is not the intention of the debugger to disable the breakpoint, it may single step over the instruction at the breakpoint, and then replace the TRAP syllable.

  Unlike the other undefined traps (which may be used as arbitrary software traps), these traps behave like hardware debug traps. That is, they will be handled by halting the core if the core is in external debug mode (i.e. the E flag in CR_DCR is set). This means that an external debugger can also use this system to support an arbitrary number of breakpoints.

  Likewise, disabling breakpoints using the B flag in CR_CCR will prevent even the TRAP instruction from actually generating a trap.

- TRAP_STEP_COMPLETE = 0xFB

  This trap is generated by hardware whenever the S flag in CR_DCR is set while debug traps are enabled. This allows the debug system to single-step. Refer to the documentation of CR_DCR for more information.

- TRAP_HW_BREAKPOINT_0 = 0xFC
- TRAP_HW_BREAKPOINT_1 = 0xFD
- TRAP_HW_BREAKPOINT_2 = 0xFE

- TRAP_HW_BREAKPOINT_3 = 0xFF

  These traps are generated by hardware when the corresponding hardware breakpoint or watchpoint is hit while debug traps are enabled.

5.4 State saving and restoration

Upon entering a trap, it is mostly up to the software to save and restore the processor state. Specifically, the software must ensure that the state of the general purpose registers, branch registers and the link register is as it was when the trap handler was entered when the RFI instruction is executed. The hardware will handle saving and restoration of the context control flags in CR_CCR and the program counter, as both of these are modified immediately when entering the trap handler. CR_CCR is saved in and restored from CR_SCCR, the program counter is saved in and restored from CR_TP.

Aside from restoring the state of the currently running task, an operating system environment may also wish to restore the state of a different task. In this case, the complete state of a task is defined by the contents of the general purpose register file, the branch register file, the link register, the program counter (to be accessed using CR_TP) and the context control register (to be accessed using CR_SCCR).

6 Reconfiguration and sleeping

The process in which the ρ-VEX processor switches between one large core and more smaller cores is called reconfiguration. Reconfigurations may be requested by the software running on the processor or the debugging interface by writing the requested configuration to a control register. The reconfiguration controller will then temporarily stop all contexts that will be affected by the reconfiguration, commit the new configuration, and (re)start any contexts that are part of the new configuration but are currently stopped.

6.1 Configuration word encoding

A configuration is described by means of a single register at most 32-bits in size. The actual size depends on the design-time configuration of the core; in particular, the number of lane groups and the number of contexts.

In the configuration word, each nibble (group of 4 bits, represented by a single hexadecimal digit) maps to a lane group. The nibble signifies the context that is to be run on that lane group. Disabling a lane group to save power is also possible, by selecting 'context' eight. This will never map to an actual context, as the maximum amount of hardware contexts supported by the design-time configuration system is also eight, and numbering starts at zero.

Obviously, not all 4.2 billion 32-bit values represent valid configurations. Configuration words must adhere to the following rules.

- The nibbles for existing pipelane groups may be set to either zero through the number of hardware contexts minus one to select a context, or eight to disable the pipelane group. For instance, the configuration word 0x7777 is illegal on an ρ-VEX processor that does not support eight hardware contexts. Configuration words like 0x9999 are reserved for future configurations, such as fault tolerant duplicate and triplicate modes.

- The nibbles for non-existant pipelane groups must be set to zero. For instance, the configuration word 0x88880000 is illegal for an ρ-VEX processor that is design-time configured to only support 4 lane groups, even though it may make more sense than the configuration word that was probably the intention here, which is simply zero.

- Any context may only be mapped to a power-of-two of contiguous pipelane groups. For instance, configuration words 0x1118 and 0x1231 are illegal, because the mapping for context 1 violates these rules.

- A set of pipelane groups mapped to a single context must be aligned. Mathematically, the index of the first pipelane group in the set must be divisible by the

cardinality of the set. For instance, the configuration word 0x0112 is illegal, because the mapping for context 1 is improperly aligned.

The reconfiguration controller will ensure that a configuration word is valid before committing it to the processor. If an invalid configuration is requested, the E flag in `CR_GSR` is set and the request is otherwise ignored.

6.2 Requesting a reconfiguration

There are three ways in which a reconfiguration can be requested.

- Writing to the `CR_CRR` context control register from a program running on the core. This section primarily deals with this mechanism.
- Writing to the `CR_BCRR` global control register from the debug bus. This mechanism is equivalent to the first, except it is triggered from outside the core.
- Using the sleep and wake-up system, as described in Section sec:core-ug-reconf-saw.

Usually, when a reconfiguration is requested, the new configuration will be committed within something in the order of tens of cycles, depending on how long it takes the reconfiguration controller to pause the affected contexts. However, a reconfiguration may also be rejected, either another context or the bus is requesting a new configuration simultaneously and arbitration is lost, or because the requested configuration is invalid. The following C function correctly deals with arbitration, and performs a best-effort attempt at detecting errors without using locks implemented in software.

```
/**
 * Requests a reconfiguration. Returns 1 if reconfiguration was successful,
 * -1 if the requested configuration is invalid or 0 if it is not known
 * whether the configuration was valid or not.
 */
int reconfigure(unsigned int newConfiguration) {

    // Extract our own context ID from the register file, which we will use
    // to check if we won arbitration or not.
    int ourselves = CR_CID;

    // Used to store the ID of the winning context after the request.
    int winner;

    // Retry requesting the new configuration until we win arbitration.
    do {

        // Request the new configuration.
        CR_CRR = newConfiguration.

        // Load the GSR register for state information.
        gsr = CR_GSR;

        // Extract the reconfiguration requester ID field from GSR.
```

```
        int winner = (gsr & CR_GSR_RID_MASK) >> CR_GSR_RID_BIT;

    } while (winner != ourselves);

    // Busy-wait for reconfiguration to complete.
    while (gsr & CR_GSR_B_MASK) {
        gsr = CR_GSR;
    }

    // If our context is still the one that was the last to request a
    // reconfiguration, the error flag in GSR is also meant for us. If not,
    // there is no way to tell if the configuration we requested was valid
    // or not.
    if (((gsr & CR_GSR_RID_MASK) >> CR_GSR_RID_BIT) != ourselves) {
        return 0;
    }

    // If the error flag is set, return -1.
    if (gsr & CR_GSR_E_MASK) {
        return -1;
    }

    // Reconfiguration was successful.
    return 1;

}
```

6.3 Sleep and wake-up system

The sleep and wake-up system refers to two context control registers that only exist on context zero, through which the processor can be set up to automatically request a reconfiguration when the interrupt request input of context zero is asserted. More specifically, the wakeup system will activate when all of the following conditions are met.

- The S flag in `CR_SAWC` is set.
- An interrupt is pending on context 0.
- Context 0 is not already active in the current configuration.
- There is no reconfiguration in progress.

When activated, the following actions are performed.

- A reconfiguration to the configuration stored in `CR_WCFG` is requested.
- `CR_WCFG` is set to the old configuration.
- The S flag in `CR_SAWC` is cleared.

This system may be used to save power that is otherwise wasted in an idle loop, or to improve interrupt latency by dedicating hardware context zero to only handling interrupts. These use cases are described below.

6.3.1 Power saving

To conserve power, the user may want to switch to a configuration where all pipelane groups are idle until an interrupt occurs. This is called sleeping. On an FPGA this is merely a proof of concept, but in an ASIC the amount of power that might be saved by clock gating or powering down the computational resources may be very significant. To go to sleep, the program should take the following steps.

1. If other hardware contexts were running other tasks in parallel to context zero, which may be in a state in which the processor should not sleep, first request these tasks to pause gracefully. If necessary, request a reconfiguration to configuration zero, as described in Section 6.2. to disable all contexts except for context zero.

2. Disable interrupts using the I field in `CR_CCR`.

3. If necessary, ensure that no interrupt occured before interrupts were disabled that should cause the processor to stay awake. If this did happen, take the appropriate actions, such as re-enabling interrupts, before attempting to sleep again.

4. Copy `CR_CC`, the current configuration, to `CR_WCFG`, the wake-up configuration. This is an easy way to ensure that `CR_WCFG` will not contain an invalid configuration. Writing to `CR_WCFG` also sets the S flag in `CR_SAWC` to enable the wake-up system.

5. Request a reconfiguration to the configuration where all pipelane groups are disabled, for instance 0x8888 on a core that is design-time configured to have four pipelane groups, as described in Section 6.2.

6. Busy-loop until the S flag in `CR_SAWC` is cleared. This ensures that the program will not continue until after the processor has finished sleeping.

7. Enable interrupts using the I field in `CR_CCR` to service the interrupt. The fact that this is not done automatically also allows the interrupt request input to simply be used as a wake-up input in a simple system where no interrupts exist.

6.3.2 Decreasing interrupt latency

To decrease interrupt latency, context zero may be used as a dedicated context for servicing interrupts. This prevents the context zero trap handler from having to save and restore the state of the processor as it was before the interrupt trap, as this information is not relevant. The other hardware contexts may be used to run the main program; the reconfiguration system is then used for hardware context switching.

To initialize this system, the program should do the following in context zero.

1. Set up links to the trap and panic handlers for context 0 in `CR_TH` and `CR_PH`.

2. Copy `CR_CC`, the current configuration, to `CR_WCFG`, the wake-up configuration. This is an easy way to ensure that `CR_WCFG` will not contain an invalid configuration. Writing to `CR_WCFG` also sets the S flag in `CR_SAWC` to enable the wake-up system.

3. Request a reconfiguration as described in Section 6.2, to, for instance, 0x1111, if the main program is to run in hardware context 1.
4. Busy-loop until the S flag in CR_SAWC is cleared. This ensures that the program will not continue until after the first interrupt is requested.
5. Set ready-for-trap and enable interrupts using the R and I fields in CR_CCR to service the interrupt.
6. Busy-loop forever to wait for the interrupt to be serviced.

The other contexts can initialize in the usual manner. The context 0 trap handler should do the following.

1. Perform body of the regular trap handling tasks, i.e., everything except for saving and restoring the context and executing RFI.
2. Set ready-for-trap and enable interrupts using the R and I fields in CR_CCR to quickly service the next interrupt if one is already pending. Clear ready-for-trap and disable interrupts in the next cycle again; one cycle is enough for an interrupt to be handled.
3. Store the contents of CR_WCFG in a temporary register.
4. Copy CR_CC, the current configuration, to CR_WCFG, the wake-up configuration. This is an easy way to ensure that CR_WCFG will not contain an invalid configuration. Writing to CR_WCFG also sets the S flag in CR_SAWC to enable the wake-up system.
5. Request a reconfiguration to the configuration as stored in the temporary register, as described in Section 6.2.
6. Busy-loop until the S flag in CR_SAWC is cleared. This ensures that the program will not continue until after the first interrupt is requested.
7. Set ready-for-trap and enable interrupts using the R and I fields in CR_CCR to service the interrupt.
8. Busy-loop forever to wait for the interrupt to be serviced.

7 Debugging ρ-VEX software

There are two main approaches to debugging the ρ-VEX processor. This chapter documents the external debugger approach. In this approach, a computer is connected to the ρ-VEX is used to debug the processor and the software running on it. The computer is connected to the ρ-VEX using some interface, usually a serial port in the case of the ρ-VEX. The alternative approach is called self-hosted debug, where the debugger runs on the ρ-VEX itself in order to debug another thread. However, this approach requires a sophisticated multithreading operating system, such as a Linux kernel with the `ptrace` system call implemented for the ρ-VEX. Although the hardware should be ready for such a system, the software for it has not yet been implemented.

7.1 Setting up

The connection between the computer and the ρ-VEX is called the debug link. Currently, the following options exist.

- A serial port, through the ρ-VEX debug support peripheral.
- PCI express, developed in [8].
- Memory mapped on a Zynq FPGA, running Linaro Linux with `rvsrv` on the embedded ARM processor. The debug commands may be given in Linaro, or `rvd` can connect to the Zynq development board using ethernet.

Which connections are supported depends on the platform. The serial port option is available in all hardware platforms except for `zed-almarvi`. The PCI express connection is supported in addition to the the serial link by `ml605-grlib` to allow faster memory access. `zed-almarvi` only supports the memory-mapped option.

Whichever platform you use, you need to execute the following commands in a console from the root directory of the platform you are using to set up the debugging environment.

```
make debug
source debug
```

The first command generates a script called 'debug' that sets up environment variables to allow you to use `rvd`. The second command runs that script. The next step depends on whether the FPGA board is connected to your machine (Section 7.3) or to another machine (Section 7.2). In the latter case, you need to be able to `ssh` to that machine.

7.2 Connecting to a remote machine

To connect to the remote machine, we will use `ssh` to forward two TCP/IP ports. You can do this by running the following command in a second terminal (you will need to keep it running), obviously replacing `<user@host>` with the computer you are connecting to and your account name on that computer.

```
ssh -N -L 21078:localhost:21078 -L 21079:localhost:21079 <user@host>
```

Note that you will *not* drop to a terminal on the remote computer as `ssh` normally does. It will appear like it is not doing anything after requesting your password (if required). You can test the connection by running `rvd ?` in the original terminal. If that does not crash with the message `Failed to connect to rvsrv`, you are ready to move on to Section 7.4. Otherwise, `ssh` is not working, or more likely, `rvsrv` is not running on the remote machine. In the latter case, you can try to start it yourself by `ssh`'ing to the machine normally and following the steps in 7.3. If that does not work, you will have to ask the owner of the machine for help.

7.3 Connecting to the FPGA

This section assumes that you are using a serial port debug link. The PCI express connection is more complicated to set up due to the drivers required. If you are using the Zedboard, refer to the separate documentation in the `zed-almarvi` platform.

If this is the first time you are connecting to the FPGA, open the following file in a text editor.

```
<rvex-rewrite>/tools/debug-interface/configuration.cfg
```

If this file does not exist, create it by copying `default-configuration.cfg` from the `src` directory. This file describes the interfaces that the debug server (`rvsrv`) will connect to or expose. The relevant configuration key is `SERIAL_PORT`, which needs to be set to the `tty` corresponding to the serial port.

When that has been configured, the debug server can be started in the terminal in which we have sourced the `debug` script using the following command.

```
make server
```

You can now test the connection to the ρ-VEX by running `rvd ?`.

7.4 Running programs

The procedure for uploading and running a program differs from platform to platform, but usually, the following three commands will work.

```
make upload-<program>
make start-<program>
make run-<program>
```

The difference between them is that `upload` only uploads the program to the ρ-VEX without starting it, `start` uploads and then starts the program, and `run` also waits for completion and prints the performance counter values. Usually, running `make` without parameters will (among other things) print a list of the available programs.

7.5 Debugging programs

The standard and recommended way to send debug commands to the ρ-VEX is to use `rvd`. All documentation for using `rvd` is embedded inside the program: just run `rvd help`. To get command specific documentation, use `rvd help <command>`.

`rvd` has builtin commands for halting, resuming, single stepping, resetting execution and printing the current state of the processor, in addition to the raw memory access commands. More complicated things, such as breakpoints, need to be set manually by accessing the control registers of the ρ-VEX. You do not have to remember the control register addresses by heart though; you can use the control register names without `CR_` prefix directly.

`rvd` has a concept of contexts. By default, the debug interface for context 0 is used. To select a different context, you can either specify the context using the `-c` command line parameter (for example, `rvd -c3 resume`) or you can set it for future commands using the `rvd select` command. In addition to specifying a single context, you can also specify a range of contexts (`<from>..<to>` or all contexts (`all`). When more than one context is selected, `rvd` will simply execute the given command for all selected contexts sequentially.

An alternative to `rvd`'s interface, the `gdb` port can be used. In this case, the following command should be used.

```
rvd gdb -- <path_to_gdb> [parameters passed to gdb]
```

This runs `gdb` as a child process to `rvd`. The appropriate parameters are passed to `gdb` to have it connect to `rvd` using the remote serial protocol, in addition to the parameters specified on the command line. A description of how to use the ρ-VEX `gdb` port is beyond the scope of this manual.

7.6 Tracing execution

The ρ-VEX can be configured at design time to include a trace unit. This allows the hardware to output a stream of data describing everything that the processor is doing at various levels of detail. `rvd` supports tracing using the following command.

```
rvd trace <output_file> [level_of_detail] [condition]
```

When executed, `rvd` writes the specified level of detail or 1 by default to the trace control byte in `CR_DCR2`. It then resumes execution on the selected contexts and reads data from the trace buffer. Tracing stops when the specified condition evaluates to 0, or when no more data is available if no condition is specified.

Terminating a trace with `ctrl+c` is not recommended, because it prevents `rvd` from resetting the trace control byte and emptying the trace buffer. To terminate a trace

gracefully when no condition is specified and the program is stuck in a loop, run `rvd break` in a separate terminal. This will make `rvd trace` assume that the program has finished executing.

`rvd trace` dumps the raw trace data to a file. This file can be converted to a human readable format using the `rvtrace` tool. If a disassembly file generated using `objdump -d` is specified in addition to the binary trace file, the disassembled instructions will be included in the trace output file.

Please note that the human readable trace files are much larger than the binary data format. It may thus take some time and a lot of disk space to generated the human readable file. You may want to pipe the output of `rvtrace` to `less` instead, so the output will only be saved in memory.

8 Design-time configuration

The ρ-VEX core is design-time configured by means of two different systems: the VHDL generics passed to the toplevel core entity and the configuration scripts.

VHDL generics

VHDL generics are used to configure the most important metrics of the core, such as the issue width, the degree of reconfigurability, the functional unit distribution, and complexity of the debug support system. Refer to Section 9.2.2.1 for more information about what exactly the generics control.

Because the generics are specified per instantiation of the core, it is possible to have differently configured ρ-VEX cores in a single design. This allows for heterogeneous multicore systems.

The values of the generics are represented as read-only registers in the global control register file in a generic way. The registers are designed such that future additions to the core are unlikely to require restructuring the existing registers, making them forward compatible. In addition, they are structured such that it is easy to extract information from the data, usually even by visually inspecting the hexadecimal values. The global control registers are described in detail in Section 4.1.

Configuration scripts

The 'configuration scripts' refer to a set of Python scripts residing in the `config` directory in the root of the ρ-VEX repository. When run by calling `make` in the root of the `config` directory, these scripts read a set of configuration and template files, to generate various sources in the repository. These sources vary from key VHDL sources for the ρ-VEX core, to memory map headers for `rvd` and the build system, to the LaTeX source files for this very document. The philosophy is that this not only makes it easier to change key components of the core, but that it should also stimulate developers to keep the documentation up-to-date, without the primary source for documentation needing to be comments in the VHDL sources.

The configuration scripts control the following processor features.

- Global and context control register file functionality, memory map and documentation (Section 8.1).
- Instruction set encoding and documentation, as well as assembly syntax (Section 8.2).
- Pipeline configuration of the ρ-VEX core (Section 8.3).

- Trap decoding and documentation (Section 8.4).

Each feature is controlled by a set of LaTeX-like files and/or key-value configuration files. These LaTeX-like files are not intended to be processed by anything other than the scripts — they cannot be processed by LaTeX directly. The only reason for their syntax to be derived from LaTeX is because it allows the documentation sections to be properly syntax-highlighted.

Each class of configuration files supports a set of non-standard commands that define the configuration. These commands are described in the following sections, as referenced in the list.

As stated, the configuration needs to be manually committed to the repository by calling `make` in the `config` directory. Changing the configuration files without doing this has no effect. This command also regenerates this PDF file, but it does not rebuild or test anything else. It is highly advised to run the conformance test suite in `platform/core-tests` after changing the configuration.

8.1 Control register files

The control register file configuration files reside in the `config/cregs` directory of the ρ-VEX repository. The configuration consists of a set of LaTeX-styled files, interpreted ordered alphabetically by their filenames. The configuration controls roughly the following things.

- The address of each control register, within hardcoded limits. The global register file is mapped from `0x000` to `0x100`, whereas the context control register file is mapped from `0x200` to `0x400`.
- The documentation for each control register, as it appears in Sections 4.1 and 4.2 of this manual.
- The functionality of each register, described using a special C-like language, which may be compiled to VHDL and C. The latter is intended for a cycle-accurate simulator, but this does not exist yet.
- The VHDL entity interface of the `cxreg` and `gbreg`, such that the implementations of the registers can communicate with the rest of the processor. If the interface is changed, the instantiation of `cxreg` and `gbreg` in `core.vhd` must be changed accordingly to make the connections.

The first of the following sections describes the LaTeX-style commands that are recognized by the configuration scripts. Any other commands are interpreted as being part of the LaTeX documentation sections. The remaining sections document the 'language-agnostic' mini-language used to describe the register logic implementations. This language-agnostic code can be transformed by the configuration scripts into both VHDL for the hardware and C for a simulator, although the latter is not yet utilized.

8.1.1 `.tex` command reference

The following LaTeX-like commands are interpreted by the Python scripts to define the control registers. They must be the only thing on a certain line aside from optional LaTeX-style comments at the end of the line, otherwise they are interpreted as part of a documentation section.

```
- \contextInterface{}, \globalInterface{}
   '- \ifaceGroup{title}
       '- \ifaceSubGroup{}
           |- \ifaceIn{unit}{name}{type}
           |- \ifaceOut{unit}{name}{type}{expr}
           |- \ifaceInCtxt{unit}{name}{type}
           '- \ifaceOutCtxt{unit}{name}{type}{expr}

- \defineTemplate{name}{parameter list}

- \register{mnemonic}{name}{offset},
  \registergen{python range}{mnemonic}{name}{offset}{stride}
   '- \field{range}{mnemonic}
       |- \reset{bit vector}
       |- \signed{}
       |- \id{identifier}
       |- \declaration{}
       |   |- \declRegister{name}{type}{expr}
       |   |- \declVariable{name}{type}{expr}
       |   '- \declConstant{name}{type}{expr}
       |
       |- \implementation{}
       |- \resetImplementation{}
       |- \finally{}
       '- \connect{output}{expr}

- \perfCounter{mnemonic}{name}{offset}
   |- \declaration{}
   |   |- \declRegister{name}{type}{expr}
   |   |- \declVariable{name}{type}{expr}
   |   '- \declConstant{name}{type}{expr}
   |
   '- \implementation{}
```

`\contextInterface {}`

`\globalInterface {}`

These commands describe the port map of the context register logic and the global register logic respectively. They may appear more than once in the configuration; their contents will simply be appended.

`\ifaceGroup {<title>}`

`\ifaceSubGroup {}`

These commands define port groups for code readability. The toplevel group has a title. Both group commands will interpret the text following the command as comments for the code.

`\ifaceIn {<unit>}{<name>}{<type>}`

\ifaceOut {<unit>}{<name>}{<type>}{<expr>}

\ifaceInCtxt {<unit>}{<name>}{<type>}

\ifaceOutCtxt {<unit>}{<name>}{<type>}{<expr>}

These commands define ports. For inputs, the signal name will be `<unit>2cxreg_<name>` or `<unit>2gbreg_<name>`. For outputs it will be `cxreg2<unit>_<name>` or `gbreg2<unit>_<name>`. `<type>` is a type specification as defined later. `<expr>` is an expression using only predefined constants (Section 8.1.4), input signals and literals. The command name determines whether the port is per-context or global and whether it is an input or output.

\register {<mnemonic>}{<name>}{<offset>}

This command starts a new register description. `<name>` is the title of the section. `<mnemonic>` is the mnemonic of the register, excluding the `CR_` prefix. The mnemonic must be mix of up to eight uppercase, number or underscore characters, and must be unique. The register may be referenced in LaTeX as `\creg {<mnemonic>}`; this will generate a hyperlink in the PDF to the register documentation. `<offset>` should be a hex number starting with `0x` divisible by 4, representing the byte offset from the control registers base. Global registers should be within the `0x000..0x0FF` range, context registers should be within `0x200..0x3FF`. `0x100..0x1FF` is reserved for the general purpose register file.

\registergen {<python range>}{<mnemonic>}{<name>}{<offset>}{<stride>}

Same as `\register` , but specifies a list of registers. `<python range>` is executed as a Python expression, expected to generate an iterable of integers. A register is generated for each of these iterations. The offset for each register is computed as `<offset>` $+iter*$ `<stride>`. `\n {}` expands to the number when used inline in `<mnemonic>` and `<name>`. In the documentation it expands to `$n$`.

\field {<range>}{<mnemonic>}

This command defines a field in the current register. A range specification is either a single bit index for a single-bit field, or of the form `<from>..<to>`, where `<from>` is the higher bit index, and both the `<from>` and `<to>` bits are included in the range. For example, `3..1` includes bits 1, 2 and 3. `<mnemonic>` should be a short, uppercase identifier for the field, which must be unique within the register. It should be as short as possible, in particular for single-bit fields, as it needs to fit in the layout of the documentation. It also needs to be a valid C and VHDL identifier, so for instance spaces and hyphens are not allowed.

\reset {<bit vector>}

This command sets the reset state of the previously defined field. If not specified, the reset state is assumed to be zero. The number of characters in `<bit vector>` must equal the number of bits in the field.

\signed {}

This command marks a field as being a signed number. The default is unsigned.

\id {<identifier>}

This command gives a field an alternative name for the C/VHDL/`rvd` definitions. This only works for 8-bit and 16-bit fields that are properly aligned.

\declaration {}

This command specifies the local register, variable and constant declaration section for this field implementation.

\declRegister {<name>}{<type>}{<expr>}

\declVariable {<name>}{<type>}{<expr>}

\declConstant {<name>}{<type>}{<expr>}

These commands specify registers, variables or constants respectively. These may be used by the implementation code. `<name>` must start with an underscore, and will expand to `cr_<register-mnemonic>_<field-mnemonic><name>`. It must be a valid C and VHDL identifier. `<type>` is a type name, as defined in Section 8.1.3. `<expr>` is an expression which defines the constant, initial value or reset value, using only predefined constants (Section 8.1.4) for a constant value, and only inputs or predefined constants for variables and registers.

\implementation {}

This command starts a language-agnostic code section as defined in Section 8.1.2, executed every rising clock edge with `clkEn` high and `reset` inactive. The following variables are predefined to interface with the core and debug busses for regular register fields.

- `_write`: the value being written if the corresponding `_wmask` bits are set.
- `_wmask`: write mask for each bit in the field, honoring both writes from the core directly and writes from the debug bus.
- `_wmask_dbg`: same as `_wmask`, but only honors debug bus accesses.
- `_wmask_core`: same as `_wmask`, but only honors accesses made by the core directly.
- `_read`: this must be written to in the `\implementation {}` section to specify the read value for the field.

The types of these variables are `bitvec`s with the same width as the field. As an example of how to use these, a simple register may be created as follows. This requires `_reg` to be declared using `\declRegister` as a `bitvec` of the field size.

```
_reg = (_reg & ~_wmask) | (_write & _wmask);
_read = _reg;
```

Performance counter implementations do not have these variables. They have `_add` instead. This is a `byte`-typed variable which specifies how much should be added to the performance counter in this cycle. The bus interfacing logic is generated to conform to Section 4.3.

\resetImplementation {}

This command starts a language-agnostic code section as defined in Section 8.1.2, executed every rising clock edge with `clkEn` high, while the global `reset` signal is inactive but the context-specific reset signal is active. This allows register implementations to override a soft context reset, for instance to make register values persistent in this case. This is necessary for, for instance, the B flag in `CR_DCR`, to allow the debugger to reset the core without immediately starting execution. This command is only allowed for context-specific registers.

\finally {}
This command starts a language-agnostic code section as defined in Section 8.1.2, executed every rising clock edge with `clkEn` high and `reset` inactive, after all `\implementation {}` sections have been processed. Variables from other fields and registers may be read in this section, in addition to all the objects which are accessible from `\implementation {}`. This allows registers to be written combinatorially from multiple field implementations, by having the regular field implementations prepare variables that describe the new value, and subsequently combining the variable values in a `\finally {}` section.

\connect {<output>}{<expr>}
This command combinatorially connects the specified output port with the specified expression. This expression may use registers and predefined constants (Section 8.1.4). This may be used to easily connect an output port to an internal register. Note however, that since inputs cannot be used in the expression, it still cannot make a combinatorial path from an input to an output. This is illegal specifically because such combinatorial paths would be needlessly difficult to model with a simulator.

\perfCounter {<mnemonic>}{<name>}{<offset>}
This command generates a performance counter register conforming to Section 4.3. The counter will occupy two 32-bit register slots from `<offset>` onwards, holding up to 7 bytes worth of counter data. The implementation expects `_add` to be set to the value which is to be added to the counter; all the bus interfacing logic is generated. The counter value register is accessible from other implementations as `CR_<mnemonic>_<mnemonic>0_r`.

8.1.2 Language-agnostic code (LAC) sections

The 'language-agnostic code' sections define the behavior of control registers. Language-agnostic code, or LAC, is a C-like domain-specific language developed specifically for describing registers in the ρ-VEX. The configuration scripts are capable of transforming LAC into both VHDL and C with relative ease. The latter is intended for a cycle-accurate simulator, but at the time of writing this simulator does not exist yet.

As LAC is C-like, LaTeX-style comments cannot coexist with it, due to the C `%` operator for modulo. Because of this, C-styled comments are used within the LAC sections. In order to prevent confusion with syntax-highlighting editors and ambiguity about where the section ends, LAC sections may be enclosed by `\begin {lstlisting}` and `\end {lstlisting}` tags. LaTeX syntax highlighters should disable highlighting in these blocks.

8.1.3 LAC type system

The primitive types supported by LAC and their VHDL equivalents are shown in Table 8.1. The C equivalents of the types range from `uint8_t` to `uint64_t`. The smallest available C type that the LAC/VHDL type fits in is used. Note that this means that `bitvec` and `unsigned` types of more than 64 bits are not supported.

In addition to the scalar primitives in the table, LAC also supports hardcoded aggregate types, i.e. the equivalent of a VHDL record or C struct. Rudimentary support is provided for array-typed aggregate members to be compatible with existing VHDL

Table 8.1: LAC primitive types.

LAC type name	Supported values	VHDL type name
boolean	true or false	boolean
natural	0..2147483647 (31 bit)	natural
bit	'0' or '1'	std_logic
bitvec<n>	n bits of '0' or '1'	std_logic_vector(n-1 downto 0)
unsigned<n>	n bits of '0' or '1'	unsigned(n-1 downto 0)

data structures at the time it was developed, though these arrays can only be indexed by integer literals.

In addition, objects can be instantiated per hardware context, which also results in an array. If per-context objects are used in a context control register implementation, they are implicitely indexed by the context which the register belongs to. Otherwise, a context may only be explicitely specified as an integer literal.

Arrays present a problem in VHDL code output. Of the primitive types, only bit actually has a VHDL array type. To get around this, and also to have the generated code be consistent with the human-written VHDL sources, a number of derived types are available. These are listed along with the supported aggregate types in Table 8.2.

Table 8.2: LAC derived types.

LAC	VHDL	C
byte (bitvec8)	rvex_byte_type rvex_byte_array	uint8_t
data (bitvec32)	rvex_data_type rvex_data_array	uint32_t
address (bitvec32)	rvex_address_type rvex_address_array	uint32_t
sylstatus (bitvec16)	rvex_sylStatus_type rvex_sylStatus_array	uint16_t
brregdata (bitvec8)	rvex_brRegData_type rvex_brRegData_array	uint8_t
trapcause (bitvec8)	rvex_trap_type rvex_trap_array	uint8_t
twobit (bitvec2)	rvex_2bit_type rvex_2bit_array	uint8_t
threebit (bitvec3)	rvex_3bit_type rvex_3bit_array	uint8_t
fourbit (bitvec4)	rvex_4bit_type rvex_4bit_array	uint8_t
sevenByte (bitvec56)	rvex_7byte_type rvex_7byte_array	uint64_t
trapinfo (aggregate)	trap_info_type trap_info_array	trapInfo_t
breakpointinfo (aggregate)	cxreg2pl_breakpoint_info_type cxreg2pl_breakpoint_info_array	breakpointInfo_t
cachestatus (aggregate)	rvex_cacheStatus_type rvex_cacheStatus_array	cacheStatus_t
cfgvect (aggregate)	rvex_generic_config_type -	cfgVect_t

Coercion and typecasts

Typically, LAC will take care of typing for you by coercing one type into another on the fly. If LAC does not know how to do it or a cast would be ambiguous, you can cast manually using C-style typecasts. The following rules apply.

- Conversions between `boolean` and `natural` works as they do in C. That is, `false` converts to zero and vice versa, `true` converts to one, and nonzero converts to `true`.
- Conversions between `boolean` and `bit` use positive logic. That is, `'1'` equals `true` and `'0'` equals `false`.
- `bit` and `bitvec1` are interchangeable.
- `bitvec<n>` and `unsigned<n>` are interchangeable.
- When a `bitvec<n>` is cast to a `bitvec` of different size, the vector is zero-extended or truncated.
- When a `bitvec<n>` is cast to a `natural` or vice versa, the value is zero-extended or truncated, with the `natural` behaving as a 31-bit value.

Access types

Aside from having a type that describes what kind of values are allowed for an object, LAC objects also have an 'access type'. This describes the access priviliges, scoping rules and general behavior of an object. The following access types are available.

- *Input*: represents an input port of the VHDL entity. They are available everywhere but in constant object initializers. They are read only.
- *Register output*: represents an output port of the VHDL entity, driven from the clocked process. They are write only.
- *Combinatorial output*: represents an output port of the VHDL entity, driven combinatorially using a `\connect` command. These objects may not be used in LAC sections.
- *Register*: represents a user defined register, declared using a `\declRegister` command. These may be read and written in any LAC section, regardless of where they are declared. They behave like VHDL signals; that is, when they are written to, the read value of a register is not affected until the next clock cycle.
- *Variable*: represents a user defined variable, declared using a `\declVariable` command. These may be read and written in the `\implementation` section of only the field to which they belong. Furthermore, they may be read in any `\finally` section.
- *Constant*: represents a user defined constant, declared using a `\declConstant` command. They are read only and globally scoped.

- *Predefined constant*: represents a predefined constant, such as a package constant or the `CFG` generic. They are read only and globally scoped. Section 8.1.4 lists the available predefined constants.

8.1.4 LAC predefined constants

The following predefined constants are available. These are hardcoded into the Python scripts.

- In context control register code only: `natural ctxt`. Represents the context to which the current register belongs.
- `cfgvect CFG` maps to the `CFG` generic.
- `bitvec65 RVEX_CORE_TAG` maps to the core version tag, to be stored in `CR_CTAG`.
- `natural BRANCH_OFFS_SHIFT` maps to the package constant of the same name, representing the way in which the branch offset field of instructions is encoded.
- `natural S_*` and `natural L_*` map to the pipeline stage and latency definitions defined in `core_pipeline_pkg.vhd`.

8.1.5 LAC literals

LAC supports the following literal syntaxes for literals for each primitive type.

- `boolean`: the literals for a `boolean` are `true` and `false`.
- `natural`: natural numbers can be represented in decimal, hexadecimal by prefixing `0x`, octal by prefixing `0` and binary by prefixing `0b`.
- `bit`: the literals for a `bit` are `'1'` and `'0'`.
- `bitvec`: `bitvec` literals can be represented as a binary string enclosed in double quotes, for instance `"0101"`. In addition, hexadecimal notation is supported by prefixing an `X`, for instance `X"DEADBEEF"`.
- `unsigned`: `unsigned` literals can be represented in the same way as `bitvec` literals by simply prefixing the literal with a `U`. For instance, `U"0101"` and `UX"DEADBEEF"`.

Aggregate types can be assigned and initialized using an compound literal. The syntax is similar to VHDL aggregates.

```
some_aggregate = {
  <name>                        => <expression>,
  <name>{<array index literal>} => <expression>,
  <name>{others}                => <expression>,
  others                        => <expression>
};
```

Unlike in VHDL, 'others' can assign any kind of combination of types, as long as the expression can be coerced to each member type. For instance, the aggregate others => 0 will initialize any aggregate which does not itself contain another aggregate to all zeros.

Aggregate literals can not be used anywhere in an expression like the other kinds of literals. They can only be assigned directly to an object or used as an initialization expression.

8.1.6 LAC object objects and references

All objects except for predefined constants (Section 8.1.4) need to be declared using the `\iface *` and `\decl *` commands. These objects can then be referenced as follows, for both reads and writes.

- The most basic way to reference an object is to use its full name. For inputs, this takes the form `<unit>2cxreg_<name>` or `<unit>2gbreg_<name>`, depending on whether it is part of the context or global register file interface. Likewise, outputs take the form `cxreg2<unit>_<name>` or `gbreg2<unit>_<name>`. Finally, objects declared using the `\decl *` commands take the form `cr_<register-mnemonic>_<field-mnemonic>_<name>`. The register and field mnemonics are included to make them unique within global scope.

- In the LAC sections for a certain field, objects that are declared in the same field can be referenced using just `_<name>`. That is, the `cr_<register-mnemonic>_<field-mnemonic>` part of the object name is implicit in the reference.

- To explicitly specify a context for context-specific objects, an `@` symbol and a natural literal determining the context may be appended. For instance, `_pc@2` references the value of `_pc` for context 2. Note that the context *must* be a literal, not even constant objects are permitted. The only thing that is permitted is `\n {}` in generated registers, as this expands to a natural literal before parsing, similar to how a C macro would behave. If the context is not explicitly specified, the current context is used. If there is no current context, the context must be explicitly specified.

- Member access for aggregate types is done by appending a . (decimal point) followed by the member name. Rudimentary support is provided for array members using `{<index>}`. As with explicitly specified contexts, only natural literals and `\n {}` (in a `\registergen` environment) are allowed for specifying the index.

- VHDL-like bit indexes and slices are supported for `unsigned` and `bitvec` types. Indexing a single bit is done by appending `[<position>]` to the end of the reference, where `<position>` may be any natural-typed expression. The result of the indexing operation is a `bit`. Slicing is done using `[<position>, <length>]`, where `<position>` still accepts any natural-typed expression, but `<length>` only supports natural literals. This is necessary to determine the resulting type at compile time, which is an `unsigned` or `bitvec` of size `<length>`. `<position>` specifies the lower bit index of the slice range.

8.1.7 LAC operators

Table 8.3 lists all the operators that are available in LAC. Note that member access, member array indexing, the explicit context `@` symbol and slices are not considered to be operators, but parts of a reference.

LAC operator precedence is identical to C operator precedence. As can be expected, parentheses can be used to explicitly specify evaluation order.

Table 8.3: LAC operators.

Prec.	Op.	Description	Assoc.

1	`!`
	`~`
	`(type)`
		Logical complement
		One's complement
		Type cast
			Right-to-left

2	`*`
	`/`
	`%`
		Multiplication
		Division
		Modulo
			Left-to-right

3	`+`
	`-`
	`$`
		Addition
		Subtraction
		`bitvec`/`unsigned` concatenation
			Left-to-right

4	`«` `»`	Left shift Right shift	Left-to-right

5	`<`
	`<=`
	`>`
	`>=`
		Less than
		Less than or equal to
		Greater than
		Greater than or equal to
			Left-to-right

6	`==` `!=`	Equality Inequality	Left-to-right
7	`&`	Bitwise and	Left-to-right
8	`^`	Bitwise xor	Left-to-right
9	`|`	Bitwise or	Left-to-right
10	`&&`	Boolean and	Left-to-right
11	`^^`	Boolean xor	Left-to-right
12	`||`	Boolean or	Left-to-right

8.1.8 LAC statements

LAC only supports the following statements.

`<reference> = <expression>;`
Assignment statement.

`if (<expression>) <statement>`
Conditional statement without `else`.

`if (<expression>) <statement> else <statement>`
Conditional statement with `else`.

`{ <statement*> }`
C-style block statement.

`<?vhdl ... ?>`

<?c ... ?>
Verbatim block statements. Anything written in place of the ellipsis is in principle outputted straight to the VHDL or C output. This allows the usage of constructs unknown to LAC. Even in these sections however, it is possible to have the code generator convert LAC-style references to the target language. This is particularly useful for C output, where the LAC objects are part of special data structures. The syntaxes for such converted references are as follows.

```
@read <name>
@read <name>@<context>
@lvalue <name>
@lvalue <name>@<context>
```

In addition to being convenient syntactic sugar, the LAC generator keeps track of which objects are read from and written to. Not using this syntax may result in incorrect optimizations.

8.2 Instruction set

The instruction set configuration files reside in the `config/opcodes` directory of the ρ-VEX repository. The configuration consists of a set of LaTeX-styled files, interpreted ordered alphabetically by their filenames, and a single key-value configuration file (`encoding.ini`), containing miscellaneous information for the instruction decoder. The complete configuration controls roughly the following things.

- The opcode for each syllable.
- The functionality of each syllable, by means of specifying the control signals. The functional units themselves are *not* controlled by this configuration.
- The assembly syntax for each syllable.
- Documentation for each syllable, as it appears in Section 3.7.
- The encoding of the branch offset field.

The next section describes the structure of the LaTeX-style configuration files, and the subsequent section provides a command reference. The last section describes the key-value configuration file.

8.2.1 `.tex` file structure

Every description of a syllable/opcode starts with a `\syllable` command. Any unrecognized commands or textual lines following a `\syllable` command are considered to be LaTeX documentation for the syllable. To provide structure among the many instructions, `\section` commands are used to group syllables. There is only one level of hierarchy this way (i.e. there is no `\subsection` etc.), and it must be used. That is, `\syllable` commands before the first `\section` command are illegal. Any unrecognized

command or text between a `\section` and `\syllable` command is considered to be LaTeX documentation for the syllable group.

All commands other than `\section` and `\syllable` specify attributes for the syllables. These are used to describe the characteristics and functionality of the instructions. These may be placed anywhere in the configuration files; their position relative to the `\section` and `\syllable` commands determine to which syllables they apply.

- Attribute commands placed before the first `\section` commands apply to all syllables. They can be thought of as being the default attributes. All syllable attributes *must* have a default value.
- Attribute commands placed between a `\section` and `\syllable` command apply to all syllables in the group, overriding the global defaults.
- Attribute commands placed after a `\syllable` command apply to that syllable, overriding the global and group defaults.

8.2.2 `.tex` command reference

The following LaTeX-like commands are interpreted by the Python scripts to define the instruction set. They must be the only thing on a certain line aside from optional LaTeX-style comments at the end of the line, otherwise they are interpreted as part of a documentation section.

`\section {<name>}`
This command starts a group of syllable definitions. `<name>` will appear as a section header in the documentation. Any unrecognized command or text between `\section` and the first `\syllable` command is interpreted as LaTeX documentation for the group.

`\syllable {<opcode>}{<mnemonic>}{<syntax>}`
This command starts the definition of a syllable.

- `<opcode>` should be a 9-bit binary string, used by the hardware to identify the syllable. Dashes may be used for don't cares. The 9 bits map to syllable bit 31..23. The value of the LSB is not really part of the opcode (the opcode field is only 8 bits wide), but defines whether the instruction can be used with only a register for the second operand (0), only an immediate for the second operand (1), or both (-). This bit is referred to as the `imm_sw` (immediate switch) bit.
- `<mnemonic>` is the name of the syllable. It will be made lowercase for the assembler syntax and uppercase for the documentation.
- `<syntax>` describes the assembler syntax of the syllable. In this, the LaTeX-like commands from Table 8.4 may be used inline. The `{}` may be omitted here.

Any textual lines between `\syllable` and `\section` or the next `\syllable` is interpreted as LaTeX documentation for the syllable. These text sections may also use the inline commands from Table 8.4, but here the `{}` may *not* be omitted.

Table 8.4: Mapping commands from assembly syntax to instruction encoding.

Command	In docs.	Description	Encoding	Condition
`\rd {}`	`$r0.d`	Integer destination register	Bit 22..17	–
`\rx {}`	`$r0.x`	Integer operand 1 register	Bit 16..11	–
`\ry {}`	`$r0.y`	Integer operand 2 register	Bit 10..5	`imm_sw` = 0
	`imm`	Integer immediate	Bit 10..2	`imm_sw` = 1
`\rs {}`	`$r0.1`	Stack pointer	–	–
`\bd {}`	`$r0.bd`	Branch destination register	Bit 19..17	`brFmt` = 0
			Bit 4..2	`brFmt` = 1
`\bs {}`	`$r0.bs`	Branch operand register	Bit 4..2	`brFmt` = 0
			Bit 26..24	`brFmt` = 1
`\lr {}`	`$l0.0`	Link register	–	–
`\of {}`	`offs`	Branch offset immediate	Bit 23..5	–
`\sa {}`	`stackadj`	Stack adjustment immediate	Bit 23..5	–
`\lt {}`	`tgt`	Long immediate lane target	Bit 27..25	–
`\li {}`	`imm`	Long immediate	Bit 24..2	–

`\class {<name>}`

This command specifies the resource class. `<name>` must be `ALU`, `MEM`, `MUL`, `BR` or `LIMMH`.

`\datapath {<key>}{<value>}`

`\alu {<key>}{<value>}`

`\branch {<key>}{<value>}`

`\memory {<key>}{<value>}`

`\multiplier {<key>}{<value>}`

These commands specify the control signals for the syllable. Which keys and values are recognized depend on the VHDL code in `core_opcode*_pkg.vhd`. They are documented in the comments of the code. Note that the configuration scripts do *not* perform error checking. Instruction set configuration errors thus will not appear until the VHDL code is compiled. Of course, if you are making changes here, you should test the core anyway.

`\noasm {}`

This attribute specifies that this syllable cannot appear in user-written assembly code. This is the case for LIMMH instructions, which are inferred by the assembler.

8.2.3 `encoding.ini` reference

This file currently defines only a single value. It determines which encoding is used for the relative branch offset, and may be set to 2 or 3. When set to 3, the LSB of the branch offset has a weight of to 64 bits. When set to 2, the branch offset is shifted right by one bit, to allow branching to 32 bit boundaries. The syntax of the file is shown below.

```
[encoding]
branch_offset_shift = 3
```

8.3 Pipeline

The pipeline configuration consists of a single key-value file in the `config/pipeline` directory of the ρ-VEX repository. The configuration describes how the ρ-VEX pipeline is organized by specifying the first stage and latencies of a multitude of configurable blocks, such as register read and writeback, branch unit determination, PC+1 computation, etc. The file is self-documenting by means of its comments.

8.4 Traps

The trap configuration files reside in the `config/traps` directory of the ρ-VEX repository. The configuration consists of a set of LaTeX-styled files, interpreted ordered alphabetically by their filenames. The configuration controls roughly the following things.

- The trap names and numeric identifiers.
- Decoding signals for each trap; debug and interrupt traps are handled differently by the processor.
- A pretty-printing macro for each trap.
- Documentation for each trap, as it appears in Section 5.3.

The next section describes the structure of the LaTeX-style configuration files. The subsequent section provides a command reference.

8.4.1 `.tex` file structure

The `\trap` and `\trapgen` commands start the definition of a trap or a number of similar traps respectively. Any unrecognized command or text following such a command is interpreted as being LaTeX documentation for the trap. The obligatory `\description` command defines the formatting string used to pretty-print the trap information. The remaining commands are optional decoding attributes for the traps.

8.4.2 `.tex` command reference

The following LaTeX-like commands are interpreted by the Python scripts to define the traps. They must be the only thing on a certain line aside from optional LaTeX-style comments at the end of the line, otherwise they are interpreted as part of a documentation section.

`\trap {<index>}{<mnemonic>}{<name>}`
The command starts a trap description. `<index>` is the trap index, which may range from 1 to 255. `<mnemonic>` is the trap identifier, which must be a valid C and VHDL identifier and should be uppercase. It is prefixed with `RVEX_TRAP_` in the header files. `<name>` is the LaTeX-formatted friendly name of the trap, used as the section title in the documentation.

`\trapgen {<python range>}{<start index>}{<mnemonic>}{<name>}`
This command works the same as `\trap` , but specifies a list of traps. `<python range>` is executed as a Python expression, expected to generate an iterable of integers. A trap specification is generated for each of these iterations. The index for each trap is computed as `<offset>`+*iter*. `\n {}` expands to the iterator value when used inline in `<mnemonic>` and `<name>`, as well as in `\description {<desc>}` below. In the documentation it expands to `$n$`.

`\description {<desc>}`
This command defines a formatting string used to pretty-print the trap information. It is used by the debug systems to allow the user to quickly identify the trap. In this description, the following commands may be used inline.

- `\at {}` expands to " `at <trap point>`" if the trap point is known, or to nothing if the trap point is not known. The trap point is expressed in hexadecimal notation.
- `\arg {u}` expands to the trap argument in unsigned decimal notation.
- `\arg {s}` expands to the trap argument in signed decimal notation.
- `\arg {x}` expands to the trap argument in hexadecimal notation.

`\debug {}`
Marks that this trap is a debug trap. The `{}` is required.

`\interrupt {}`
Marks that this trap is an interrupt trap. The `{}` is required.

9 Instantiation

This section describes how the ρ-VEX core and processing systems should be instantiated, what the functions of all the external signals are, and what generics are available. The first section lists the basic signal data types that will be used throughout the interfaces. The remaining sections document instantiation of the bare ρ-VEX processor core and two processing systems that incorporate the processor and local memories or cache, one that does not depend on GRLIB and one which does.

9.1 Data types

The following basic VHDL data types are used for the ports and generics. They are defined in `common_pkg`.

```
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);
subtype rvex_byte_type        is std_logic_vector( 7 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;
type rvex_byte_array          is array (natural range <>) of rvex_byte_type;
```

The `address`, `data` and `syllable` types all represent 32-bit words. The distinction is made only for clarity; one can not simply give the ρ-VEX processor 64-bit address map by widening the address type.

The `mask` type is used for byte-masking the data vectors for bus operations. As all memory operations operate on 32-bit words, the `mask` type has four bits to mask each byte. The most significant bit of the these masks maps to the most significant byte of the 32-bit word, and thus to the lowest byte address, as the ρ-VEX system is big endian.

The `byte` type should be self-explanatory.

9.2 Bare ρ-VEX processor

This section describes how the bare ρ-VEX core should be instantiated. It is intended for HDL designers who wish to design their own processing system.

9.2.1 Instantiation template

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

If you get errors when instantiating the core with this template, the documentation might be out of date. Fear not, for the signals are also documented in the entity description in `core.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
      -- ...
    ),
    CORE_ID                     => CORE_ID,
    PLATFORM_TAG                => PLATFORM_TAG

  )
  port map (

    -- System control.
    reset                       => reset,
    resetOut                    => resetOut,
    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,
    rctrl2rv_resetVect          => rctrl2rv_resetVect,
    rv2rctrl_done               => rv2rctrl_done,

    -- Common memory interface.
    rv2mem_decouple             => rv2mem_decouple,
    mem2rv_blockReconfig        => mem2rv_blockReconfig,
    mem2rv_stallIn              => mem2rv_stallIn,
    rv2mem_stallOut             => rv2mem_stallOut,
    mem2rv_cacheStatus          => mem2rv_cacheStatus,

    -- Instruction memory interface.
    rv2imem_PCs                 => rv2imem_PCs,
```

```
    rv2imem_fetch               => rv2imem_fetch,
    rv2imem_cancel              => rv2imem_cancel,
    imem2rv_instr               => imem2rv_instr,
    imem2rv_affinity            => imem2rv_affinity,
    imem2rv_busFault            => imem2rv_busFault,

    -- 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_ifaceFault          => dmem2rv_ifaceFault,
    dmem2rv_busFault            => dmem2rv_busFault,

    -- 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,

    -- Trace interface.
    rv2trsink_push              => rv2trsink_push,
    rv2trsink_data              => rv2trsink_data,
    rv2trsink_end               => rv2trsink_end,
    trsink2rv_busy              => trsink2rv_busy

  );
```

9.2.2 Interface description

As you can see in the template, the generics and signals are grouped by their function. The following subsections will document each group.

9.2.2.1 Core configuration

These generics parameterize the core.

- `CFG : rvex_generic_config_type`

  This generic contains the configuration parameters for the core. `rvex_generic_config_type` is a `record` type with the following members.

  - `numLanesLog2 : natural`

    This parameter specifies the binary logarithm of the number of lanes to instantiate. The range of acceptable values is 0 through 4, although only 1, 2 and 3 are tested. The default is 3, which specifies an 8-way ρ-VEX processor.

  - `numLaneGroupsLog2 : natural`

This parameter specifies the binary logarithm of the number of lane groups to instantiate. Each lane group can be disabled individually to save power, operate on its own, or work together on a single thread with other lane groups. May not be greater than 3 (due to configuration register size limits) or `numLanesLog2`. It is only tested up to `numLanesLog2`-1. The default is 2, specifying 4 lane groups.

— `numContextsLog2 : natural`

This parameter specifies the binary logarithm of the number of hardware contexts in the core. May not be greater than 3 due to configuration register size limits. The default is 2, specifying 4 hardware contexts.

— `genBundleSizeLog2 : natural`

This parameter specifies the binary logarithm of the number of syllables in a generic binary bundle. When a branch address is not aligned to this and `limmhFromPreviousPair` is set, then special actions will be taken to ensure that the relevant syllables preceding the trap point are fetched before operation resumes. The default is 3, specifying 8-way generic binary bundles.

— `bundleAlignLog2 : natural`

The *ρ*-VEX processor will assume (and enforce) that the start addresses of bundles are aligned to the specified amount of syllables. When this is less than `numLanesLog2`, the stop bit system is enabled. The value may not be greater than `numLanesLog2`. The default is 3, disabling the stop bit system.

— `multiplierLanes : natural`

This parameter defines what lanes have a multiplier. Bit 0 of this number maps to the first lane, bit 1 to the second lane, etc. The default is `0xFF`, specifying that each lane has a multiplier.

— `memLaneRevIndex : natural`

This parameter specifies the lane index for the memory unit, counting down from the last lane in each lane group. So `memLaneRevIndex` = 0 results in the memory unit being in the last lane in each group, `memLaneRevIndex` = 1 results in it being in the second to last lane, etc. The default is 1.

— `numBreakpoints : natural`

This parameter specifies how many hardware breakpoints are instantiated. The maximum is 4 due to the register map only having space for 4. The default is also 4.

— `forwarding : boolean`

This parameter specifies whether or not register forwarding logic should be instantiated. With forwarding disabled, the core will use less area and might run at higher frequencies, but much more NOPs are necessary between data-dependent instructions. The forwarding logic is enabled by default.

— `limmhFromNeighbor : boolean`

When this parameter is true, syllables can borrow long immediates from the neighboring syllable in a syllable pair. This is enabled by default.

— `limmhFromPreviousPair : boolean`

When this parameter is true, syllables can borrow long immediates from the previous syllable pair. This is enabled by default. This is not supported when stop bits are enabled, i.e. when `bundleAlignLog2` < `numLanesLog2`. Therefore, when stop bits are enabled, this should be disabled.

— `reg63isLink : boolean`

When this parameter is true, general purpose register 63 maps directly to the link register. When false, `MOVTL`, `MOVFL`, `STW` and `LDW` must be used to access the link register, but an additional general purpose register is available. This exists for compatibility with the ST200 series processors. It is disabled by default.

— `cregStartAddress : rvex_address_type`

This paramater specifies the start address of the 1kiB control register file as seen from the processor. It must be aligned to a 1kiB boundary. The core is not able to access data memory in the specified region. The default value is `0xFFFFFC00`, i.e. the block from `0xFFFFFC00` to `0xFFFFFFFF`.

— `resetVectors : rvex_address_array(7 downto 0)`

This parameter specifies the reset address for each context, if not overruled at runtime by connecting the optional `rctrl2rv_resetVect` signal. When less than eight contexts are instantiated, the higher indexed values are unused. The default is 0 for all contexts.

— `unifiedStall : boolean`

When this parameter is true, the stall signals for each group will be connected to the same signal. That is, if one lane group has to stall, all lane groups necessarily have to stall. This may be a requirement of the memory subsystem connected to the core; when this is enabled, the memory architecture can be made simpler, but cannot make use of the possible performance gain due to being able to stall only part of the core. This parameter is disabled by default, meaning that the stall signals are independent.

— `gpRegImpl : natural`

This parameter specifies the general purpose register implementation to use. The following values are accepted.

* `RVEX_GPREG_IMPL_MEM` (default): block RAM + LVT implementation for FPGAs.
* `RVEX_GPREG_IMPL_SIMPLE`: behavioral implementation for Synopsis.

— `traceEnable : boolean`

This parameter specifies whether the trace unit should be instantiated. It is disabled by default.

— `perfCountSize : natural`

This parameter specifies the size of the performance counters in bytes. Up to 7 bytes are supported. The default is 4 bytes.

  - `cachePerfCountEnable : boolean`

    This parameter enables or disables the cache performance counters. When enabled, the number of lane groups must equal the number of contexts, because the signals from the cache blocks are mapped to the contexts directly. In the future, the cache performance counters are to be placed in the cache instead of the core. This parameter is off by default.

  Typically, one will want to use the `rvex_cfg` function to specify this value. This function takes as its arguments values for all the record members as specified above, but has default values for each of them, meaning that not all of them have to be specified. In addition, a `base` argument of type `rvex_generic_config_type` may be specified, which will be used as the default value for unspecified parameters. This permits mutation of the `CFG` record as it passes from entity to subentity, which is otherwise impossible to do with record generics.

- `CORE_ID : natural`

  This value is used to uniquely identify this core within a multicore platform. It is made available to the programs running on the core and the debug system through `CR_COID`.

- `PLATFORM_TAG : std_logic_vector(55 downto 0)`

  This value is to uniquely identify the platform as a whole. It is intended that this value be generated by the toolchain by hashing the source files and synthesis options. It is made available to the programs running on the core and the debug system through `CR_PTAG`.

9.2.2.2 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, most components of the core will be reset, regardless of the state of `clkEn`. The only component of the core that is not reset by this is the general purpose register file. This is because this register file is implemented using block RAMs, which have no physical reset input in Xilinx FPGAs.

The `resetOut` signal is asserted high for one cycle when the debug bus writes a one to the reset bit in `CR_GSR`. This signal may be used to reset support systems as well as the core, or it may be ignored.

9.2.2.3 Run control

The run control signals provide an interface between the core and an interrupt controller or a master processor if the ρ-VEX is used as a coprocessor. 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 `clkEn`abled 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 on the subsequent clock edge. 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` before an interrupt is acknowledged may still cause an interrupt trap to be caused. This is due to the fact that traps take time to propagate through the pipeline. The core will still assert `rv2rctrl_irqAck` upon entry of the trap service routine in this case. In order to properly account for this behavior, interrupt controllers should ignore `rv2rctrl_irqAck` if no interrupt is active, and there should be a special `rctrl2rv_irqID` value that signals 'no interrupt'. The trap service routine should return to application code as soon as possible in this case.

- `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 that were already in the pipeline and have already committed data, and set the program counter to point to the next instruction that should be issued for the program to resume correctly later. As soon as `rctrl2rv_run` is asserted high again, the context will resume, assuming there is nothing else preventing it from running. When `rctrl2rv_run` is not specified, it is tied to `'1'`.

  Only when the context has completely stopped, i.e., there are no instructions in the pipeline, will `rv2rctrl_idle` be asserted high. This may also happen while `rctrl2rv_run` is high, when the core is being halted for a different reason. Such reasons include preparing for reconfiguration, the context not having lane groups assigned to it, and the B flag in `CR_DCR`. `rv2rctrl_idle` remains high until the next instruction is fetched.

- `rctrl2rv_reset : in std_logic_vector(`*number of contexts - 1* `downto 0)`
- `rctrl2rv_resetVect : in rvex_address_array(`*number of contexts - 1* `downto 0)`

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

  When `rctrl2rv_reset` is asserted high, the context control registers for the indexed context are synchronously reset in the next `clkEn`abled cycle. Note that this behavior is different from the master `reset` signal, which ignores `clkEn`. When it is not specified, it is tied to `'0'`.

  `rctrl2rv_resetVect` determines the reset vector for each context, i.e. the initial program counter. When it is not specified, it is tied to the reset vector specified by the `CFG` generic.

  `rv2rctrl_done` is connected to the D flag in `CR_DCR`, which is set when the processor executes a `STOP` instruction. The only way to clear this signal without debug bus accesses is to assert `reset` or `rctrl2rv_reset`.

  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. `rctrl2rv_resetVect` marks the entry point for the kernel.

9.2.2.4 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 PCs presented by the instruction memory ports will always be contiguous and aligned for groups that are working together. The fetch and cancel signals will always be equal.
  - The ρ-VEX assumes that the `mem2rv_blockReconfig` and `mem2rv_stallIn` signals are equal for coupled pipelane groups. Behavior is completely undefined 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 context associated with the indexed group is guaranteed to not reconfigure. The ρ-VEX will assume that the associated bits in the `mem2rv_blockReconfig` signal will always be released eventually when no operations are requested by those pipelane groups, otherwise the system may deadlock. When pipelane groups are coupled, their respective `mem2rv_blockReconfig` signals must be equal. 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, the next rising edge of the clock signal will be ignored. When pipelane groups are coupled, their respective `mem2rv_stallIn` signals must be equal. 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. In addition, the result of a previously requested memory operation should remain valid until the next `clkEn`abled cycle where the `rv2mem_stallOut` signal is low, as this is when the core will sample the signal.

  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.

- `mem2rv_cacheStatus : in rvex_cacheStatus_array(`*number of lane groups - 1* `downto 0)`

  This signal may be driven with cache status information. This is used by the trace unit only. The data type is a record defined in `core_pkg` as follows.

```
type rvex_cacheStatus_type is record
  instr_access      : std_logic;
  instr_miss        : std_logic;
  data_accessType   : std_logic_vector(1 downto 0);
  data_bypass       : std_logic;
  data_miss         : std_logic;
  data_writePending : std_logic;
end record;
```

All signals must be externally gated by the stall signals of the core for compatibility with performance counters in the future. Otherwise, the `instr_` prefixed signals share the timing of the instruction fetch result, and `data_` prefixed signals share the timing of the data memory access result.

`instr_access` should be high when an instruction fetch was performed. In this case, `instr_miss` may also be high to signal that the fetch caused a cache miss.

`data_access` should be set to `01` if a read access was performed, to `10` if a 32-bit write access was performed and to `11` if a partial write was performed. `00` logically means no operation. If an access was performed that bypassed the cache, `data_bypass` should be set. If an access was performed that caused a cache miss, `data_miss` should be set. If an access was performed by a cache block that had a nonempty write buffer when the request was made, `data_writePending` should be set.

9.2.2.5 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 without stalls 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 lane 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.

- `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 result of the most recently requested instruction fetch 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 that indexes the lane group that 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. Its value is made available to the program using the `CR_AFF` register.

- `imem2rv_busFault : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

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

9.2.2.6 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 that 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 1 kiB 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 0xFFFFFC00, meaning that addresses 0xFFFFFC00 through 0xFFFFFFFF are 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. 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 $i*8$. This corresponds to byte address $a+3-i$, where a is the word address specified by `rv2dmem_addr`, because the ρ-VEX is big endian.

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

  Active high write enable signal from the core for each memory unit. 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_ifaceFault` or `dmem2rv_busFault` 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_ifaceFault` or `dmem2rv_busFault` are high.

- `dmem2rv_ifaceFault : in std_logic_vector(`*number of lane groups - 1* `downto 0)`

  These signals are expected to be valid `L_MEM` enabled cycles after a read or write request. `dmem2rv_ifaceFault` being high indicates that the read or write could not be performed because the memory system is incapable of servicing the specific type of memory access. For instance, the reconfigurable cache asserts this signal if more than one request is made at a time by coupled lane groups. `dmem2rv_busFault` being high indicates that some kind of bus fault occured, for example if a memory access was made to unmapped memory.

  In either case, a `DMEM_FAULT` trap will be issued. The trap argument will be set to the address that was requested.

9.2.2.7 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 `clkEn`abled 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.

  The debug bus can read from and write to all ρ-VEX registers. 1024 bytes are used per context, thus the size of the debug bus control register block is $1024 \cdot numContexts$ bytes. As the upper address bits are simply ignored, this block is mirrored across the full 32-bit address space.

  The memory map of an ρ-VEX with two contexts is shown in Table 9.1. Note that the mappings per context equal those of direct accesses to the control registers from the ρ-VEX memory units (Section 3.2.4), with the addition of the general purpose registers. Additional contexts specified at design time simply appear after the first two.

Table 9.1: Debug bus memory map for 2 contexts.

Address	**Mapping**
`0x000-0x0FF`	Global control registers
`0x100-0x1FF`	Context 0 general purpose registers
`0x200-0x3FF`	Context 0 control registers
`0x400-0x4FF`	Mirror of global control registers
`0x500-0x5FF`	Context 1 general purpose registers
`0x600-0x7FF`	Context 1 control registers

9.2.2.8 Trace interface

The trace interface provides an optional write-only bus to some memory system or peripheral, which the core may send trace information to. The trace system is disabled by default and must be enabled in the `CR_DCR2` control register. In addition, the trace unit hardware is only instantiated when `traceEnable` is set in the `CFG` vector.

- `rv2trsink_push : out std_logic`

  When high, `rv2trsink_data` and `rv2trsink_end` are valid and should be registered in the next cycle where `clkEn` is high.

- `rv2trsink_data : out rvex_byte_type`

  Trace data signal. Valid when `rv2trsink_push` is high.

- `rv2trsink_end : out std_logic`

When high, this is the last byte of this trace packet. May be used to flush buffers downstream, or may be ignored.

- `trsink2rv_busy : in std_logic`

  When high while `rv2trsink_push` is high, the trace unit is stalled. While stalled, `rv2trsink_push` will stay high and `rv2trsink_data` and `rv2trsink_end` will remain stable.

9.3 Standalone processing system

The ρ-VEX standalone processing system has the following features.

- Single cycle local instruction memory implemented in block RAMs.
- Local data memory implemented in block RAMs that is single cycle for up to two accesses at a time.
- The initial contents of the local memories can be set.
- Optionally, the cache can be instantiated. In this case, a unified instruction/data memory is instantiated in block RAMs. The access latency of this memory is configurable at runtime to mimic a more realistic memory access latency for cache tests.
- An external bus for peripherals or other memories may be connected through a bus master interface. Without the cache, the ρ-VEX cannot read instructions from this bus, but it can access it using memory operations.
- A slave bus interface allows access to the ρ-VEX debug port, a trace buffer, and the local memories, as well as the cache control register if the cache is instantiated.
- The cache, if instantiated, is coherent only for accesses made by the ρ-VEX itself. A cache flush is required using the cache control register if the debug bus is used to write to the local memories.

9.3.1 Instantiation template

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

If you get errors when instantiating the core with this template, the documentation might be out of date. Fear not, for the signals are also documented in the entity description in `rvsys_standalone.vhd`.

```
library rvex;
use rvex.common_pkg.all;
use rvex.bus_pkg.all;
use rvex.bus_addrConv_pkg.all;
use rvex.core_pkg.all;
use rvex.cache_pkg.all;
```

```
use rvex.rvsys_standalone_pkg.all;

-- ...

rvex_standalone_inst: entity rvex.rvsys_standalone
  generic map (

    -- System configuration.
    CFG => rvex_sa_cfg(
      core => rvex_cfg(
        numLanesLog2              => 3,
        numLaneGroupsLog2         => 2,
        numContextsLog2           => 2
        -- ...
      ),
      core_valid => true
      -- ...
    ),
    CORE_ID                     => CORE_ID,
    PLATFORM_TAG                => PLATFORM_TAG,
    MEM_INIT                    => MEM_INIT

  )
  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,
    rctrl2rv_resetVect          => rctrl2rv_resetVect,
    rv2rctrl_done               => rv2rctrl_done,

    -- Peripheral interface.
    rvsa2bus                    => rvsa2bus,
    bus2rvsa                    => bus2rvsa,

    -- Debug interface.
    debug2rvsa                  => debug2rvsa,
    rvsa2debug                  => rvsa2debug

  );
```

9.3.2 Interface description

As you can see in the template, the generics and signals are grouped by their function. The following subsections will document each group.

9.3.2.1 System configuration

These generics parameterize the system.

- `CFG : rvex_sa_generic_config_type`

  This generic contains the configuration parameters for the core. `rvex_sa_generic_config_type` is a `record` type with the following members.

  - `core : rvex_generic_config_type`

    This parameter specifies the ρ-VEX core configuration as passed to the bare ρ-VEX processor core. Refer to Section 9.2.2.1 for more information.

  - `cache_enable : boolean`

    This parameter selects whether or not the cache should be instantiated. This is false by default.

  - `cache_config : cache_generic_config_type`

    This parameter specifies the size of the cache blocks. `cache_generic_config_type` is a record type with two `natural`-typed members: `instrCacheLinesLog2` and `dataCacheLinesLog2`. The sizes are determined as follows.

    $$\text{Instr. cache size} = 4 \cdot N_{lanes} \cdot 2^{instrCacheLinesLog2} \cdot N_{laneGroups}$$

    $$\text{Data cache size} = 4 \cdot 2^{dataCacheLinesLog2} \cdot N_{laneGroups}$$

    The number of lane groups is part of the equation because the number of lines are specified per block, and a different block is instantiated for each lane group.

  - `cache_bypassRange : addrRange_type`

    This parameter specifies the range of addresses for which the cache (if instantiated) is bypassed. This range is `0x80000000..0xFFFFFFFF` by default.`addrRange_type` is a record containing four `rvex_address_type` members: `low`, `high`, `mask`, and `match`. An address is considered to be part of the range if the following VHDL expression is true.

    ```
    unsigned(addr and mask) >= unsigned(low) and
    unsigned(addr and mask) <= unsigned(high) and
    std_match(addr, match)
    ```

    This record may be set using the `addrRange` function, which allows parameters to be omitted. The defaults for each parameter specify the complete 32-bit address range, so it is usually sufficient to only set one or two of the parameters.

  - `imemDepthLog2B : natural`

  - `dmemDepthLog2B : natural`

    These parameters specify the sizes of the local instruction and data memories respectively if the cache is not used. Otherwise, `dmemDepthLog2B` specifies the size of the unified memory and `imemDepthLog2B` is ignored. The size is specified as the logarithm of the number of bytes. The default value is 16 for both of these, resulting in 64 kiB memories.

- `traceDepthLog2B : natural`

  This parameter specifies the size of the trace buffer in the same way that the memory sizes are specified. The default value is 13, resulting in a trace buffer 8 kiB in size. This size is required if the serial debug interface is to be used, due to the way in which bulk data transfers are implemented in the serial protocol.

- `debugBusMap_imem : addrRangeAndMapping_type`
- `debugBusMap_dmem : addrRangeAndMapping_type`
- `debugBusMap_rvex : addrRangeAndMapping_type`
- `debugBusMap_trace : addrRangeAndMapping_type`

  These parameters specify which addresses on the debug bus are mapped to which device. These parameters may be specified with the `addrRangeAndMap` function, which takes the same parameters as the `addrRange` function discussed for `cache_bypassRange`. In addition, it also allows the designer to change how the address bits are mapped from source to peripheral address. Refer to the comments in `bus_addrConv_pkg.vhd` for more information.

  By default, the instruction memory is mapped to `0x10000000..0x1FFFFFFF` and to `0x30000000..0x3FFFFFFF`, the data memory is mapped to `0x20000000..0x3FFFFFFF`, the ρ-VEX debug port is mapped to `0xF0000000..0xFFFFFFFF` and the trace buffer is mapped to `0xE0000000..0xEFFFFFFF`. Note that the range `0x30000000..0x3FFFFFFF` maps to both the instruction and data memories. This range allows the instruction and data memory to be written simultaneously, limiting the upload time using the debug unit.

- `debugBusMap_mutex : boolean`

  This parameter specifies whether logic needed to handle overlaps in the debug bus address map is to be instantiated. If it is set to false, this logic is instantiated, allowing bus write commands to access multiple memories at the same time. This is the default. If it is set to true, overlaps are not supported, but a some area may be saved.

- `rvexDataMap_dmem : addrRangeAndMapping_type`
- `rvexDataMap_bus : addrRangeAndMapping_type`

  These parameters specify where data accesses from the ρ-VEX are to be routed. They work the same way as the `debugBusMap` parameters. By default, the lower half of the address space, `0x00000000..0x7FFFFFFF`, is mapped to the data memory, and the remainder is mapped to the bus. Overlaps are not allowed. Accesses made to unmapped addresses cause a bus fault.

Typically, one will want to use the `rvex_sa_cfg` function to specify this value. This function takes as its arguments values for all the record members as specified above, but has default values for each of them, meaning that not all of them have to be specified. In addition, a `base` argument of type `rvex_generic_config_type` may be specified, which will be used as the default value for unspecified parameters. This

permits mutation of the CFG record as it passes from entity to subentity, which is otherwise impossible to do with record generics.

Important note: in order to allow the function to detect whether the core and cache_config fields are specified, the core_valid and cache_config_valid parameters must be set to true, or the defaults will be substituted!

- CORE_ID : natural

  This value is used to uniquely identify this core within a multicore platform. It is made available to the programs running on the core and the debug system through CR_COID.

- PLATFORM_TAG : std_logic_vector(55 downto 0)

  This value is to uniquely identify the platform as a whole. It is intended that this value be generated by the toolchain by hashing the source files and synthesis options. It is made available to the programs running on the core and the debug system through CR_PTAG.

- MEM_INIT : rvex_data_array

  This value is used to initialize the instruction and data memories. If left unspecified, the memories are initialized to zero.

9.3.2.2 System and run control interfaces

These interfaces are identical to those specified for the bare ρ-VEX core in Sections 9.2.2.2 and 9.2.2.3.

9.3.2.3 Peripheral and debug interfaces

- rvsa2bus : out bus_mst2slv_type
- bus2rvsa : in bus_slv2mst_type

  These signals form a master ρ-VEX bus device, allowing the ρ-VEX to access memory or peripherals outside the processing system. A number of bus interconnection primitives are available in rvex_rewrite/lib/rvex/bus. Instantiation of these primitives is beyond the scope of this manual.

- debug2rvsa : in bus_mst2slv_type
- rvsa2debug : out bus_slv2mst_type

  These signals form a slave ρ-VEX bus device, allowing devices outside the processing system, such as the debug serial port peripheral, to access the local memories, trace buffer and the ρ-VEX control registers.

  The memory map of the debug interface is specified using generics. If the cache is instantiated, The cache control register is mapped to the same address as CR_AFF. Because CR_AFF is read-only and the cache control register is write only, this does not cause conflicts. The cache control register has the following layout.

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
LAT		DFL	IFL

LAT field, bits 31..24

Must be written to a value between 1 and 254 inclusive for correct operation. That amount of cycles plus one are added to the bus access delay in case of a cache bypass, write or miss.

DFL field, bits 15..8

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the data cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect.

IFL field, bits 7..0

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the instruction cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect.

9.4 GRLIB processing system

The ρ-VEX GRLIB-based processing system has the following features.

- One AHB master interface per ρ-VEX lane group.
- Cache snooping on the AHB bus guarantees cache coherency with other processors and the debug interface sharing the same bus.
- A LEON3 interrupt controller compatible interface is exposed. This allows the ρ-VEX to use the interrupt controller that comes with GRLIB.
- For simulation, an S-record file specifying the expected memory contents can be specified. Every instruction fetch and data access made by the ρ-VEX is snooped and checked against this memory. The memory automatically updates when the ρ-VEX writes a value. Whenever the cache returns an unexpected or inconsistent value, a VHDL warning is printed.

9.4.1 Instantiation template

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

If you get errors when instantiating the core with this template, the documentation might be out of date. Fear not, for the signals are also documented in the entity description in `rvsys_standalone.vhd`.

```
library rvex;
use rvex.common_pkg.all;
use rvex.bus_pkg.all;
use rvex.bus_addrConv_pkg.all;
```

```
use rvex.core_pkg.all;
use rvex.cache_pkg.all;
use rvex.rvsys_grlib_pkg.all;

library grlib;
use grlib.amba.all;
use grlib.devices.all;

library gaisler;
use gaisler.leon3.all;

-- ...

rvex_grlib_inst: entity rvex.rvsys_grlib
  generic map (

    -- System configuration.
    CFG => rvex_grlib_cfg(
      core => rvex_cfg(
        numLanesLog2              => 3,
        numLaneGroupsLog2         => 2,
        numContextsLog2           => 2
        -- ...
      ),
      core_valid => true,
      cache => cache_cfg(
        instrCacheLinesLog2       => 18,
        dataCacheLinesLog2        => 18
      ),
      cache_valid => true
    ),
    PLATFORM_TAG                => PLATFORM_TAG,
    AHB_MASTER_INDEX_START      => RVEX_MST_INDEX,
    CHECK_MEM                   => false,
    CHECK_MEM_FILE              => ""
  )
  port map (

    -- System control.
    clki                        => clki,
    rstn                        => rstn,

    -- AHB interface.
    ahbmi                       => ahbmi,
    ahbmo                       => ahbmo_rvex,
    ahbsi                       => ahbsi,

    -- Debug interface.
    bus2dgb                     => bus2dgb,
    dbg2bus                     => dbg2bus,

    -- LEON3 compatible interrupt controller interface.
    irqi                        => irqi,
    irqo                        => irqo
```

```
);
```

9.4.2 Interface description

As you can see in the template, the generics and signals are grouped by their function. The following subsections will document each group.

9.4.2.1 System configuration

These generics parameterize the system.

- `CFG : rvex_grlib_generic_config_type`

  This generic contains the configuration parameters for the core. `rvex_grlib_generic_config_type` is a `record` type with the following members.

  - `core : rvex_generic_config_type`

    This parameter specifies the ρ-VEX core configuration as passed to the bare ρ-VEX processor core. Refer to Section 9.2.2.1 for more information.

  - `cache : cache_generic_config_type`

    This parameter specifies the size of the cache blocks. `cache_generic_config_type` is a record type with two `natural`-typed members: `instrCacheLinesLog2` and `dataCacheLinesLog2`. The sizes are determined as follows.

    $$\text{Instr. cache size} = 4 \cdot N_{lanes} \cdot 2^{instrCacheLinesLog2} \cdot N_{laneGroups}$$

    $$\text{Data cache size} = 4 \cdot 2^{dataCacheLinesLog2} \cdot N_{laneGroups}$$

    The number of lane groups is part of the equation because the number of lines are specified per block, and a different block is instantiated for each lane group.

  Similar to the bare ρ-VEX and the standalone platform, the `rvex_grlib_cfg` function is available to set this record.

  Important note: in order to allow the function to detect whether the `core` and `cache` fields are specified, the `core_valid` and `cache_valid` parameters must be set to true, or the defaults will be substituted!

- `PLATFORM_TAG : std_logic_vector(55 downto 0)`

  This value is to uniquely identify the platform as a whole. It is intended that this value be generated by the toolchain by hashing the source files and synthesis options. It is made available to the programs running on the core and the debug system through `CR_PTAG`.

- `AHB_MASTER_INDEX_START : natural`

  This value must be set to the AHB master index of the first lane group. The remaining lane groups are mapped to subsequent master indices. In addition, this value is made available to the programs running on the core and the debug system through `CR_COID`, to allow a program to uniquely identify which ρ-VEX it is running on in a multicore system.

- `CHECK_MEM : boolean`

- `CHECK_MEM_FILE : string`

  These parameters configure the simulation-only memory consistency checking system. `CHECK_MEM` enables or disables the system. If the system is enabled, `CHECK_MEM_FILE` must specify the filename of an S-record file holding the initial memory contents. The filename must be relative to the simulator search path.

9.4.2.2 System control interface

The system control signals include the clock source and the reset signal. All registers are rising-edge triggered. The reset signal is active-low to comply with the AHB standard. It is inverted in the system before it is passed to the ρ-VEX logic blocks, which assume an active-high reset signal.

9.4.2.3 AHB interface

- `ahbmi : in ahb_mst_in_type`

- `ahbmo : out ahb_mst_out_vector_type(`*number of lane groups - 1* `downto 0)`

  These signals represent the AHB master interfaces that the cache blocks use as their data source. One master interface is required for each ρ-VEX lane group. The master indices must be a contiguous range, starting at the index specified by the `AHB_MASTER_INDEX_START` generic.

- `ahbsi : in ahb_slv_in_type`

  This signal must be tied to the signal that the AHB interconnect logic broadcasts to all AHB slaves. It is used by the cache to monitor the bus for cache coherence purposes.

9.4.2.4 Debug interface

- `bus2dgb : in bus_mst2slv_type`

- `dbg2bus : out bus_slv2mst_type`

  The debug interface is a slave ρ-VEX bus device. It allows access to all control registers in the system and the trace buffer. It should be connected to the AHB bus using the `ahb2bus` bridge, possibly through additional ρ-VEX bus interconnect primitives. This allows a single AHB to ρ-VEX bus bridge to be used for multiple

ρ-VEX devices, similar to how an APB bridge allows multiple APB peripherals to share a single AHB slave interface.

The ρ-VEX bus primitives are available in `rvex_rewrite/lib/rvex/bus`. Instantiation of these primitives is beyond the scope of this manual.

The debug interface port has a fixed memory map, shown below.

Trace buffer	0x3FFF 0x2000
ρ-VEX context 7 registers	0x1FFF 0x1D00
Unused	...
ρ-VEX context 6 registers	0x1BFF 0x1900
Unused	...
ρ-VEX context 5 registers	0x17FF 0x1500
Unused	...
ρ-VEX context 4 registers	0x13FF 0x1100
Unused	...
ρ-VEX context 3 registers	0x0FFF 0x0D00
Unused	...
ρ-VEX context 2 registers	0x0BFF 0x0900
Unused	...
Cache control register	0x0803 0x0800
ρ-VEX context 1 registers	0x07FF 0x0500
Unused	...
Reset register	0x0400
ρ-VEX context 0 registers	0x03FF 0x0100
ρ-VEX global control registers	0x00FF 0x0000

The reset register resets the entire processing system when written. The only thing that is not reset is the AHB bus bridge to prevent deadlocks due to the AHB bus interconnect not being reset.

The cache control register has the following layout. Note that it is different from the standalone system cache control register.

Offset	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
0x0800	IFL	DFL		B

IFL field, bits 7..0

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the instruction cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect. The register always reads as 0.

DFL field, bits 15..8

Each of these bits corresponds to an ρ-VEX lane group. Writing a one to a bit causes the data cache block corresponding to the indexed lane group to be flushed. Writing a zero has no effect. The register always reads as 0.

B flag, bit 0

When this bit is set, the data cache is always bypassed. When it is cleared, the cache is only bypassed for memory accesses to the upper half of the address space, i.e. `0x80000000..0xFFFFFFFF`. The flag resets to 0.

9.4.2.5 Interrupt controller interface

- `irqi : in irq_in_vector(0 to` *number of contexts - 1*`)`
- `irqo : out irq_out_vector(0 to` *number of contexts - 1*`)`

  These signals should be tied to a GRLIB `irqmp` interrupt controller, with the number of processors set to the number of ρ-VEX contexts.

Bibliography

[1] J. van Straten, "A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support," Master's thesis, Delft University of Technology, the Netherlands, 2016.

[2] A. Brandon and S. Wong, "Support for Dynamic Issue Width in VLIW Processors using Generic Binaries," in *Proc. Design, Automation & Test in Europe Conference & Exhibition*, (Grenoble, France), pp. 827 – 832, March 2013.

[3] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing*. Morgan Kaufmann, 2005.

[4] Hewlett-Packard Company, "HP Labs : Downloads: VEX." Available: `http://www.hpl.hp.com/downloads/vex/` (visited on Feb. 12, 2016).

[5] P. Faraboschi, G. Brown, J. Fisher, G. Desoll, and F. Homewood, "Lx: a Technology Platform for Customizable VLIW Embedded Processing," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 203–213, June 2000.

[6] STMicroelectronics, "ST200 Micro Toolset Releases." Available: `http://ftp.stlinux.com/pub/tools/products/st200tools/index.htm` (visited on Feb. 12, 2016).

[7] R. Seedorf, "Fingerprint Verification on the VEX Processor," Master's thesis, Delft University of Technology, the Netherlands, 2010.

[8] H. van der Wijst, "An Accelerator based on the ρ-VEX Processor: an Exploration using OpenCL," Master's thesis, Delft University of Technology, the Netherlands, 2015.