ρ-VEX reference manual

Jeroen van Straten

January 18, 2016

Contents

Todo list

Introduction

<div style="text-align: right">**1**</div>

This manual is intended to be used as a reference when working with the ρ-VEX reconfigurable VLIW processor ecosystem. The initial version was written in parallel with X. Both document essentially the same thing, but there are some key differences in their intended audience and, thus, what they focus on.

> Jeroen's thesis title once one exists + reference

- The thesis is intended for the scientific reader, who is expected to be interested primarily in the new concepts introduced in this version of the ρ-VEX processor, and how they affect the system. As such, the thesis does not describe in detail the parts of the ρ-VEX core that have little research value. It will instead provide external references for supplemental information.

- This manual is intended for readers who intend to work with the ρ-VEX processor or want to add onto it. It thus strives to provide detailed documentation of all the parts of the system. In addition, external references are intentionally kept to a minimum in favor of copying information, to prevent the reader from having to constantly cross reference. Finally, the author hopes that this document will be modified over time, to reflect additions and changes made to the system. This is obviously not possible with a thesis.

While certainly possible, it is not intended that one reads this manual linearly. That would be like reading a dictionary from A to Z. Instead, the reader is advised to search for information recursively, first and foremost using the table of contents. Failing that, each section starts with a basic introduction, briefly describing the contents of its subsections. If you are reading this document digitally, references will be clickable to allow you to quickly jump to the parts you are interested in. In addition, almost all syllables, control registers and traps are clickable as well, to jump to their documentation.

1.1 Organization

The second chapter doubles as an introduction to the ρ-VEX processor architecture and a reference manual for those who intend to use the core. The third chapter documents the internal workings of the ρ-VEX processor core, intended for those who intend to modify or add to the core. The fourth chapter is similar, but instead documents the reconfigurable cache. The fifth chapter documents the bus system used within the ρ-VEX system to tie components together. The sixth chapter documents the debug support UART peripheral. The seventh chapter documents the ρ-VEX platforms, the colloquial name for the hardware systems that tie the previously described components together.

The eight chapter descibes the host software systems, focussing primarily on the build system, the simulator and the debug communication link with the hardware. Finally, ninth chapter describes the general purpose software that has been written for the ρ-VEX processor so far.

ρ-VEX core user guide

<div style="text-align: right; font-size: 3em;">**2**</div>

This chapter is intended as a user guide for using the ρ-VEX processor core. For documentation about how the core works internally, refer to Chapter 3 instead.

The first section of this chapter gives a general introduction on VLIW processor architecture and the parts that make the ρ-VEX processor special. The next section describes the instruction set architecture (ISA) in detail, including a list of all instructions and their encodings. The third section lists and documents all the control registers. The fourth section describes the trap and interrupt model of the core, and subsequently lists all currently defined traps. The fifth section briefly documents the reconfiguration system. Finally, the last section documents how the core is configured and instantiated in an HDL design.

2.1 Introduction to the ρ-VEX processor

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.

2.1.1 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 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.

---

[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.

2.1.2 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 run-time, 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*. 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.

> Reference to Anthony's generic binary paper here

2.1.3 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.

2.2 Instruction set architecture

The instruction set architecture (ISA) of the ρ-VEX processor is based on the HP VEX ISA and the ST200 processor family by STMicroelectronics . More specifically, the ISA is mostly copied from the previous version of the ρ-VEXprocessor . 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.

2.2.1 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, $r0.55
  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.

Memory references use the following syntax: *literal*[$r0.*index*]. At runtime, the literal is added to the register value to get the address.

Any literal may be a decimal or hexadecimal number (using 0x notation), a label reference, or a basic C-like integer expression.

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.

2.2.2 Registers

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

11

2.2.2.1 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 the register is not special.

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
```

2.2.2.2 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.

2.2.2.3 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.

2.2.2.4 Global and context control registers

These two register files contain special-purpose registers. The global control registers contain status information not related 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 by memory operations outside the core. In most platforms this happens coincidentally, as the processor can access the main bus of the platform, and the main bus is wired to the debug bus.

For more information about the control registers, refer to Section 2.3.

2.2.3 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 2.2.2.4 for more information.

2.2.4 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.

2.2.4.1 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.

2.2.4.2 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.

2.2.4.3 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.

2.2.4.4   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.

2.2.4.5   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 2.2 for an 8-way ρ-VEX processor.
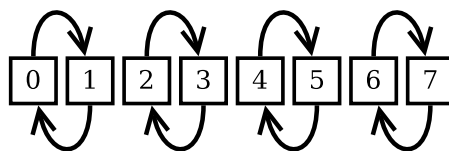

Figure 2.1: Long immediate from neighbor routing.

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 2.2 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.
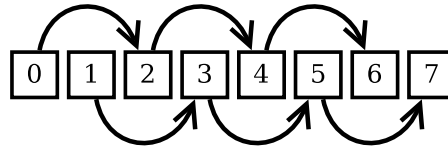

Figure 2.2: Long immediate from previous pair routing.

2.2.5   Generic binaries

This section might contain trace amounts of bullshit. Anthony should probably look over this sometime, as he knows more about it than I do. The vexparse stuff might also be wrong because I can't remember properly what I all added to it and what it could already do and I haven't looked into it yet. - Jeroen

Generic binaries are binaries that can be correctly run on different core configurations, even if the core reconfigures during execution. They were introduced in Anthony's generic binary paper. 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).

Insert reference to Anthony's generic binary paper

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.

2.2.5.1   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 Anthony's generic binary paper 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[2] 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.

Insert reference to Anthony's generic binary paper

2.2.6 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 needs to work through eight syllables.[3] When stop bits are enabled, such alignment `NOP`s

---

[2]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.

[3]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

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.

2.2.6.1 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.

2.2.7   Instruction set

The ρ-VEX instruction set consists of 169 instructions. These instructions are defined by two bitfields in the syllable, called opcode and imm_sw. The opcode field is 8 bits in size,

---

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.

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 113 `opcodes` 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 2.2.6 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]
0 0 0 1 0 0 1 1	1	d	x	imm		S	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 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	offs						S	goto offs
0 0 1 0 0 0 0 1							S	igoto $l0.0
0 0 1 0 0 0 1 0	offs						S	call $l0.0 = offs
0 0 1 0 0 0 1 1							S	icall $l0.0 = $l0.0
0 0 1 0 0 1 0 0	offs					bs	S	br $b0.bs, offs
0 0 1 0 0 1 0 1	offs					bs	S	brf $b0.bs, offs
0 0 1 0 0 1 1 0	stackadj						S	return $r0.1 = $r0.1, stackadj, $l0.0
0 0 1 0 0 1 1 1	stackadj						S	rfi $r0.1 = $r0.1, stackadj
0 0 1 0 1 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 1	0	bd	x	y			S	cmpeq $b0.bd = $r0.x, $r0.y
0 1 0 0 0 0 0 1	1	bd	x	imm			S	cmpeq $b0.bd = $r0.x, imm
0 1 0 0 0 0 1 0	0	d	x	y			S	cmpge $r0.d = $r0.x, $r0.y
0 1 0 0 0 0 1 0	1	d	x	imm			S	cmpge $r0.d = $r0.x, imm
0 1 0 0 0 0 1 1	0	bd	x	y			S	cmpge $b0.bd = $r0.x, $r0.y
0 1 0 0 0 0 1 1	1	bd	x	imm			S	cmpge $b0.bd = $r0.x, imm
0 1 0 0 0 1 0 0	0	d	x	y			S	cmpgeu $r0.d = $r0.x, $r0.y
0 1 0 0 0 1 0 0	1	d	x	imm			S	cmpgeu $r0.d = $r0.x, imm
0 1 0 0 0 1 0 1	0	bd	x	y			S	cmpgeu $b0.bd = $r0.x, $r0.y
0 1 0 0 0 1 0 1	1	bd	x	imm			S	cmpgeu $b0.bd = $r0.x, imm
0 1 0 0 0 1 1 0	0	d	x	y			S	cmpgt $r0.d = $r0.x, $r0.y
0 1 0 0 0 1 1 0	1	d	x	imm			S	cmpgt $r0.d = $r0.x, imm
0 1 0 0 0 1 1 1	0	bd	x	y			S	cmpgt $b0.bd = $r0.x, $r0.y
0 1 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

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	1	d	x	imm	S	cmpgtu $r0.d = $r0.x, imm
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 1 0	0	d	x	y	S	add $r0.d = $r0.x, $r0.y
0 1 1 0 0 0 1 0	1	d	x	imm	S	add $r0.d = $r0.x, imm
0 1 1 0 0 0 1 1	0	d	x	y	S	and $r0.d = $r0.x, $r0.y
0 1 1 0 0 0 1 1	1	d	x	imm	S	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 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
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		x	y	S	trap $r0.x, $r0.y
1 0 0 1 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

2.2.7.1 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.

```
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, used in conjunction with ADDCG.

> Figure out how divisions work so it can be explained here.

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

```
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;
```

2.2.7.2 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];
```

2.2.7.3   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];
```

2.2.7.4 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	

27

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;
```

2.2.7.5   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]);
```

2.2.7.6   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];
```

2.2.7.7 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];
```

2.2.7.8 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;
```

2.2.7.9   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.

2.2.7.10 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.

> Properly document the multiply instructions. This is a bit of a pain due to the fact that they make no sense to me at all.

```
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	

Multiply signed low 16 x low 16 bits.

```
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	

Multiply unsigned low 16 x low 16 bits.

```
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	

Multiply signed low 16 (s1) x high 16 (s2) bits.

```
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	

Multiply unsigned low 16 (s1) x high 16 (s2) bits.

```
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	

Multiply signed high 16 x high 16 bits.

```
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	

Multiply unsigned high 16 x high 16 bits.

```
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	

Multiply signed low 16 (s2) x 32 (s1) bits.

39

```
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	

Multiply unsigned low 16 (s2) x 32 (s1) bits.

```
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	

Multiply signed high 16 (s2) x 32 (s1) bits.

```
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	

Multiply unsigned high 16 (s2) x 32 (s1) bits.

```
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	

Multiply signed high 16 (s2) x 32 (s1) bits, shift left 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	

Multiply unsigned low 16 (s2) x signed 32 (s1) bits, shift right 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	

Multiply signed high 16 (s2) x 32 (s1) bits, shift right 16.

2.2.7.11 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
```

2.2.7.12 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.

2.2.7.13 Long immediate instructions

`limmh tgt, 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	0		tgt											imm														S	

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 2.2.4.5 for more information.

2.3 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 2.2.2.4.

2.3.1 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
0x0F8	COID	PTAG0	PTAG1	PTAG2	CR_PVER1
0x0FC	PTAG3	PTAG4	PTAG5	PTAG6	CR_PVER0

47

2.3.1.1 `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 2.5.2 for more information.

2.3.1.2 `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 2.5.1 and 2.5.2 for more information.

2.3.1.3 `CR_CC` - Current configuration 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	
0x008			CC		CR_CC
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 hardwired to the current configuration vector. Refer to Section 2.5.1 for more information.

2.3.1.4 `CR_AFF` - Cache affinity 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	
0x00C			AF		CR_AFF
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 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.

2.3.1.5 `CR_CNT` - Cycle counter 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	
0x010			CNT		CR_CNT
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					

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.

2.3.1.6 `CR_CNTH` - Cycle counter register high

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	
0x014		CNTH		CNT	CR_CNTH
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 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 2.3.3 for more information. Note however, that unlike the other performance counters, this register always exists, regardless of the design-time configured performance counter width.

2.3.1.7 `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 `CR_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.

2.3.1.8 `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.

2.3.1.9   `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 Core Debug	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				

This register is reserved for future expansion.

2.3.1.10   `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 Core Debug	0 0 0 0	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *	* * * *	

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.

2.3.1.11   `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.

2.3.1.12   `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.

2.3.1.13 `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.

2.3.1.14 `CR_EXT1` - Extension 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	
0x0E4					CR_EXT1
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.

2.3.1.15 `CR_EXT0` - Extension register 0

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	
0x0E8	T | BRK	C | P			O|L|F	CR_EXT0
Reset	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 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 2.3.3 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.

2.3.1.16 `CR_DCFG` - Design-time configuration 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	
0x0EC			BA	NC	NG	NL	CR_DCFG
Reset	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	* * * *	* * * *	* * * *	* * * *	
Core							
Debug							

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.

2.3.1.17 `CR_CVER1` - Core version 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	
0x0F0	VER	CTAG0	CTAG1	CTAG2	CR_CVER1
Reset	0 0 1 1 0 0 1 1	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	
Core					
Debug					

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.

2.3.1.18 `CR_CVER0` - Core version register 0

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	
0x0F4	CTAG3	CTAG4	CTAG5	CTAG6	CR_CVER0
Reset	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	
Core					
Debug					

Refer to `CR_CVER1` for more information.

2.3.1.19 `CR_PVER1` - Platform version 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	
0x0F8	COID	PTAG0	PTAG1	PTAG2	CR_PVER1
Reset	* * * * * * * *	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	
Core					
Debug					

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 multi-core 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.

2.3.1.20 `CR_PVER0` - Platform version register 0

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	
0x0FC	PTAG3	PTAG4	PTAG5	PTAG6	CR_PVER0
Reset	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	0 * * * * * * *	

Core
Debug

Refer to `CR_PVER1` for more information.

2.3.2 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

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	
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
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

2.3.2.1 `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 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.

57

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. Currently, this flag only controls whether the MMU is enabled; the ρ-VEX processor does not have any security features yet. In kernel mode, the MMU is bypassed; in user mode, it is activated. 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`.

> The kernel mode/MMU enable flag exists, but the MMU is not in the design yet.

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.

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.

2.3.2.2 `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 enables external interrupts, writing 10 disables them 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`.

2.3.2.3   `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.

2.3.2.4   `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.

2.3.2.5   `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.

2.3.2.6 `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 2.4 for more information.

2.3.2.7 `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.

2.3.2.8 `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 2.4 for more information.

2.3.2.9 `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.

2.3.2.10 `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 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.

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.

2.3.2.11 `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.

2.3.2.12   `CR_CRR` - Context 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	
0x240			CRR		CR_CRR
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 core only. When it is written, a reconfiguration is requested. Refer to Section 2.5 for more information.

2.3.2.13   `CR_WCFG` - Wakeup configuration

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	
0x248			WCFG		CR_WCFG
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 only exists on context 0. This configuration register is used in conjunction with the S flag in `CR_SAWC`. Refer to Section 2.5.3 for more information.

2.3.2.14   `CR_SAWC` - Sleep and wake-up 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	
0x24C				RUN S	CR_SAWC
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 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 2.5.3 for more information.

2.3.2.15 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.

2.3.2.16 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 are controlled by hardware context 0, 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 2.4 for more information.

2.3.2.17 `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 `CSC`n 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.

2.3.2.18 `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.

2.3.2.19 `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.

2.3.2.20 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.21 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.22 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.23 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.24 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.25 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.26 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.27 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.28 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.29 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.30 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.31 `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 2.3.3 for more information about the structure of performance counters.

2.3.2.32 `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 2.3.3 for more information about the structure of performance counters.

2.3.3 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.

```c
/**
 * 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.

2.4 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.

2.4.1 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.

2.4.2 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.

2.4.3 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 sec-

tion), 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.

76

- `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` $= 0\text{x}09$

  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` $= 0\text{xF8}$

- `TRAP_SOFT_DEBUG_1` $= 0\text{xF9}$

- `TRAP_SOFT_DEBUG_2` $= 0\text{xFA}$

  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` $= 0\text{xFB}$

  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` $= 0\text{xFC}$

- `TRAP_HW_BREAKPOINT_1` $= 0\text{xFD}$

- `TRAP_HW_BREAKPOINT_2` $= 0\text{xFE}$

- `TRAP_HW_BREAKPOINT_3` $= 0\mathrm{xFF}$

  These traps are generated by hardware when the corresponding hardware breakpoint or watchpoint is hit while debug traps are enabled.

2.4.4 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`).

2.5   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.

2.5.1   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.

2.5.2 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.

```c
/**
 * 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;

}
```

2.5.3   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 im-
prove interrupt latency by dedicating hardware context zero to only handling interrupts.
These use cases are described below.

2.5.3.1 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 2.5.2. 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 2.5.2.

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.

2.5.3.2 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 2.5.2, 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 2.5.2.

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.

Write more about the sleep and wake-up system

2.6 Configuration and instantiation

This section describes how the core should be instantiated, what the function of all the external signals are, and how the core may be design-time configured. It is intended for HDL designers who wish to incorporate the bare-metal core into their design.

2.6.1 Data types

The following basic VHDL data types are used for the ports and generics. They are defined in `common_pkg`.

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

Refer to some place that designers can turn to if they want to instantiate higher-level ρ-VEX core blocks, such as the cached core.

2.6.2 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`.

Make sure the instantiation template is at least up-to-date at the time of writing; this was copied from the manual from a year ago.

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

-- ...

rvex_inst: entity rvex.core
  generic map (

    -- Core configuration.
    CFG => rvex_cfg(
      numLanesLog2                => 3,
```

```
        numLaneGroupsLog2           => 2,
        numContextsLog2             => 2
        -- ...
    )

)
port map (

    -- System control.
    reset                           => reset,
    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

);
```

2.6.3   Port description

As you can see in the template, signals are grouped by their function. The following subsections will document each group of signals.

2.6.3.1   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.

2.6.3.2 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 `clkEnabled` cycle. `rctrl2rv_irqID` is sampled in exactly that cycle and is made available to the trap handler through the trap argument register. When not specified, `rctrl2rv_irq` is tied to `'0'` and `rctrl2rv_irqID` is tied to `X"00000000"`.

  When `rv2rctrl_irqAck` is high, an interrupt controller would typically release `rctrl2rv_irq` and set `rctrl2rv_irqID` to a value signalling that no interrupt is active 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 `clkEnabled` 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.

2.6.3.3   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 `clkEnabled` 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.

  Note that these signals are very cache implementation dependent. They were designed to work with the reconfigurable cache described in Section 4 specifically. It may, of course, be abused by other cache implementations, as long as the people working with the resulting traces are adequatly informed.

2.6.3.4 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.

2.6.3.5   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.

  > There is currently no way to distinguish between a data memory interface fault and a bus fault. A new trap should probably be added to the core for this sometime.

2.6.3.6 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 `clkEnabled` 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.

2.6.3.7 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.

2.6.4 Generic configuration

Write about the CFG generic.

2.6.5 Package configuration

The ρ-VEX processor has more configuration options than those described by the `CFG` generic. These configuration options are instead

ρ-VEX core internals

<div style="text-align: right; font-size: 3em;">**3**</div>

This chapter documents how the ρ-VEX processor core works internally. Refer to Chapter 2 instead if you are only interested in using the core as is.

The first section of this chapter gives an architectural overview of the core, lists the VHDL files that represent the core and deals with code style. The second and third sections together describe how instructions are executed; the former documents the datapath (tentatively, the lifespan of an instruction) while the latter documents how the next instruction is chosen. The fourth section deals with the reconfiguration system of the core, and what the interconnect between the lane groups and the contexts looks like. Finally, the fifth section documents the external debug and trace interfaces.

3.1 Overview

This section gives an overview of the structural hierarchy of the core, as well as a VHDL entity and package file listing. The last section documents the coding style employed within all ρ-VEX core files.

3.1.1 Architecture

A block diagram of the rvex core is shown in Figure 3.1. The abbreviations used are described in the next section. The same abbreviations are used in the code to mark the source and/or destination of a signal, as described in Section 3.1.3.
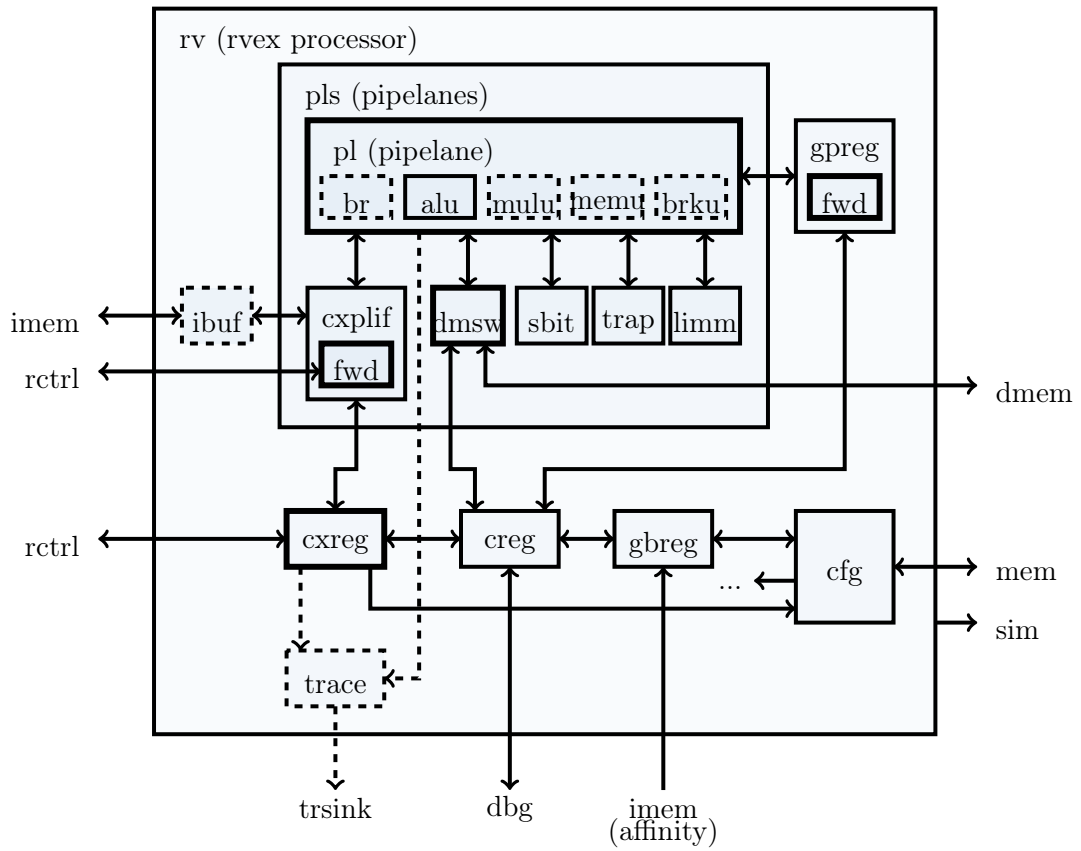
Figure 3.1: Block diagram of the ρ-VEX core.

3.1.2 File and abbreviation list

This section lists all the core files and their functions. Files describing entities also have their entity abbreviation listed. Abbreviations without a corresponding file are also listed, at the bottom of the list.

ρ-VEX processor   `rv`   core.vhd

This is the toplevel file for the ρ-VEX processor.

External ρ-VEX package   `-`   core_pkg.vhd

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

Internal ρ-VEX package   `-`   core_intIface_pkg.vhd

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

Pipelanes   `pls`   core_pipelanes.vhd

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

Pipelane   `pl`   core_pipelane.vhd

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

Pipeline configuration   `-`   core_pipeline_pkg.vhd

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

Branch unit   `br`   core_branch.vhd

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

Arithmetic logic unit   `alu`   `core_alu.vhd`

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

Memory unit   `memu`   `core_memu.vhd`

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

Multiply unit   `mulu`   `core_mulu.vhd`

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

Breakpoint unit   `brku`   `core_brku.vhd`

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

Opcode package   -   `core_opcode_pkg.vhd`

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

Datapath ctrl. signal package -   `core_opcodeDatapath_pkg.vhd`

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

ALU ctrl. signal package   -   `core_opcodeAlu_pkg.vhd`

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

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

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

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

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

Branch ctrl. signal package - `core_opcodeBranch_pkg.vhd`

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

(Dis)assembler package - `core_asDisas_pkg.vhd`

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

General purpose registers **gpreg** `core_gpRegs.vhd`

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

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

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

Behavioral gpreg spec. - `core_gpRegs_sim.vhd`

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

Forwarding logic **fwd** `core_forward.vhd`

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

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

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

Data memory switch   `dmsw`   `core_dmemSwitch.vhd`

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

Long immediate routing   `limm`   `core_limmRouting.vhd`

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

Trap routing   `trap`   `core_trapRouting.vhd`

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

Trap code package   -   `core_trap_pkg.vhd`

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

Control registers   `creg`   `core_ctrlRegs.vhd`

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

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

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

Ctrl. reg. bank - `core_ctrlRegs_bank.vhd`

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

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

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

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

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

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

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

Context register logic **cxreg** `core_contextRegLogic.vhd`

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

Global register logic **gbreg** `core_globalRegLogic.vhd`

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

Configuration control **cfg** `core_cfgCtrl.vhd`

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

Configuration decoder - `core_cfgCtrl_decode.vhd`

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

Trace unit   `trace`   `core_trace.vhd`

This optional unit (enabled through `CFG.traceEnable`) configurably selects and compresses trace information into a byte stream. While the byte stream is being written to the memory system or peripheral connected to the trace interface of the core, this unit ensures that the core is properly stalled.

Memory   `mem`   -

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

Instruction memory   `imem`   -

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

Data memory   `dmem`   -

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

Debug bus   `dbg`   -

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

Run control   `rctrl`   -

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

Simulation   `sim`   -

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

Trace data sink   `trsink`   -

Abbreviation for the external system that the trace output data is sent to.

3.1.3   Coding style

All code within the ρ-VEX packages is wrote using a consistent code style. Special attention was paid to naming conventions, as VHDL easily becomes confusing due to the large amount of signals and variables everywhere.

- All signals, entity names, package names and types use a combination of camelCase and underscores. Typically, underscores are used as a form of hierarchy separation, where the VHDL language does not otherwise allow it, and camelCase is used to indicate word boundaries within one level of hierarchy. For example, `core_ctrlRegs_bank` refers to a (register) bank for the control registers for the ρ-VEX core.

- Most signal names start with an underscore-terminated abbreviation, which indicates the source and destination entity. This identifier contains two entity abbreviation codes separated by a `2`. The entity abbreviation codes are defined at the top of `core.vhd` and are also listed in the previous section. Sometimes other abbreviations are used for signals local to one entity, which should be clear from context.

- All constant names are uppercase with underscores.

- Labels typically use underscores only, to prevent conflicts between similarly named entities and signals.

- Types use one of the following suffixes to indicate what kind of type they represent.

    - `_type`: scalar type.
    - `_array`: array type.
    - `_ptr`: access type for a scalar.
    - `_array_ptr`: access type for an array.

- Enties and packages have the same name as their filename; exactly one entity or package is defined per VHDL file. Package names end in `_pkg`. All ρ-VEX core files start with `core_` to keep their names unique within the `rvex` package, which contains a number of supporting packages as well.

- Entity descriptions must clearly document the function of every port and generic that passes it by, *especially* when the entity generates or uses the signal (as opposed to just routing it). All hope of future generations comprehending the code is lost when interfaces are not clear.

- Ports should be grouped by function or route. The groups should be made apparent in the entity descriptions using blocks so they're easy to spot.

- Entity instantiation code should include the port group names.

- If words cannot describe how the code works, ASCII art diagrams might. This may seem a bit silly, but the only way to maintain up-to-date documentation is by having the documentation right in the developer's face. A picture somewhere in some documentation folder simply will not do. This manual is already a stretch.

- Indentation is accomplished using two spaces, tabs are not used.

- The `:` symbol in declarations, and the `=>` symbol in case statements, port maps and generic maps, is generally aligned to column 33 using spaces for aesthetically pleasing code.

- Comments must wrap at column 80 for easy readability. Code should also not be too wide, although the column 80 limit is not strictly adhered to.

3.2 Datapath

Write about how the datapath works, and how `pipeline_pkg` can be used to configure it.

3.3 Flow control

Write about flow control. A lot of this has already been done, refer to the notes folder.

3.4   Reconfiguration

Write about how reconfiguration works and how contexts and lane groups are interconnected.

3.5 External debug and trace interface

Write about how the external debug and trace systems work.

Cache

$$4$$

Bus system

5

External debug support unit $\quad$ **6**

Platforms

7

Host software

8

Target software

9