ACKNOWLEDGEMENT

By utilizing this website and/or documentation, I hereby acknowledge as follows:

Effective October 1, 2012, QUALCOMM Incorporated completed a corporate reorganization in which the assets of certain of its businesses and groups, as well as the stock of certain of its direct and indirect subsidiaries, were contributed to Qualcomm Technologies, Inc. (QTI), a wholly-owned subsidiary of QUALCOMM Incorporated that was created for purposes of the reorganization.

Qualcomm Technology Licensing (QTL), the Company's patent licensing business, continues to be operated by QUALCOMM Incorporated, which continues to own the vast majority of the Company's patent portfolio. Substantially all of the Company's products and services businesses, including QCT, as well as substantially all of the Company's engineering, research and development functions, are now operated by QTI and its direct and indirect subsidiaries[1]. Neither QTI nor any of its subsidiaries has any right, power or authority to grant any licenses or other rights under or to any patents owned by QUALCOMM Incorporated.

No use of this website and/or documentation, including but not limited to the downloading of any software, programs, manuals or other materials of any kind or nature whatsoever, and no purchase or use of any products or services, grants any licenses or other rights, of any kind or nature whatsoever, under or to any patents owned by QUALCOMM Incorporated or any of its subsidiaries. A separate patent license or other similar patent-related agreement from QUALCOMM Incorporated is needed to make, have made, use, sell, import and dispose of any products or services that would infringe any patent owned by QUALCOMM Incorporated in the absence of the grant by QUALCOMM Incorporated of a patent license or other applicable rights under such patent.

Any copyright notice referencing QUALCOMM Incorporated, Qualcomm Incorporated, QUALCOMM Inc., Qualcomm Inc., Qualcomm or similar designation, and which is associated with any of the products or services businesses or the engineering, research or development groups which are now operated by QTI and its direct and indirect subsidiaries, should properly reference, and shall be read to reference, QTI.

[1] The products and services businesses, and the engineering, research and development groups, which are now operated by QTI and its subsidiaries include, but are not limited to, QCT, Qualcomm Mobile & Computing (QMC), Qualcomm Atheros (QCA), Qualcomm Internet Services (QIS), Qualcomm Government Technologies (QGOV), Corporate Research & Development, Qualcomm Corporate Engineering Services (QCES), Office of the Chief Technology Officer (OCTO), Office of the Chief Scientist (OCS), Corporate Technical Advisory Group, Global Market Development (GMD), Global Business Operations (GBO), Qualcomm Ventures, Qualcomm Life (QLife), Quest, Qualcomm Labs (QLabs), Snaptracs/QCS, Firethorn, Qualcomm MEMS Technologies (QMT), Pixtronix, Qualcomm Innovation Center (QuIC), Qualcomm iSkoot, Qualcomm Poole and Xiam.

Hexagon Virtual Machine

Specification [Draft]

80-NB419-3 Rev. A

August 4, 2011

QUALCOMM Incorporated
5775 Morehouse Drive
San Diego, CA 92121-1714
U.S.A.

Contents

Figures

Tables

Revision History

Revision	Date	Description
A	August 2011	Initial version of document.

1 Introduction

1.1 Overview

The Hexagon™ Virtual Machine (HVM) provides system programmers with an abstraction layer which enables multiple operating systems and other clients to execute concurrently on a single Hexagon processor, through the virtualization and partitioning of physical hardware resources.

For example, HVM enables the Hexagon processor to concurrently run the following software systems while enforcing strict resource protection between them:

- A fine-tuned real-time application managing special devices and services.
- A general-purpose operating system implementing a user interface and application stack.

HVM virtualizes the Hexagon processor itself as multiple virtual processors which execute concurrently, while ensuring security and quality of service.

1.2 Architecture

Figure 1-1 shows the major functional units of the HVM architecture.

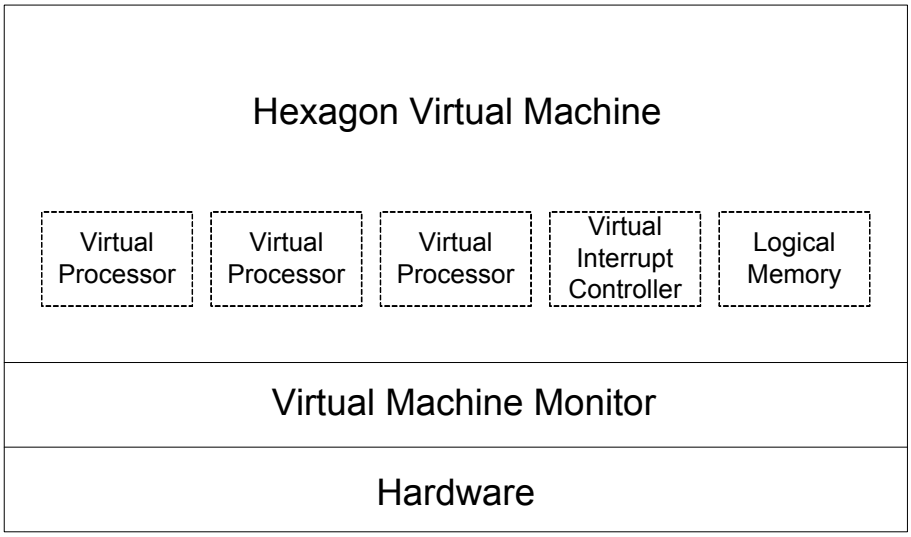

Figure 1-1 HVM architecture

The Hexagon Virtual Machine consists of the virtualized programming environment in which a client software system runs. It includes the following items:

- One or more virtual processors
- Virtual interrupt controller
- Logical memory

The virtual processors (which can be started and stopped) share the virtual interrupt controller and logical memory.

The Virtual Machine Monitor (VMM) is a software layer which manages the hardware resources and provides them to HVM in their virtualized form.

The hardware layer consists of the Hexagon hardware resources. This includes the Hexagon processor and related subsystems (memory, interrupt controller, etc.) which are managed by the VMM.

NOTE The VMM supports multiple concurrent HVM instances – each instance is referred to as a *virtual machine*.

1.3 Modes

HVM supports two levels of privilege within the virtual machine:

- HVM User mode
- HVM Guest mode

Guest mode is privileged with respect to User mode, but neither mode is privileged to directly access the underlying hardware.

1.4 Instructions and events

HVM provides virtual instructions and events which control the transitions between User and Guest modes, and which enable a guest operating system to manage the privileged virtual resources.

1.5 Virtual privileged resource model

The HVM virtual privileged resource model was designed with the following goals:

- Expose the minimum amount of detail on the underlying privileged hardware resources.
- Perform as efficiently as possible when implemented in software on the Hexagon V2 processor and later processor versions.
- Define interfaces to support their implementation in hardware (either whole or partial) on future Hexagon processor versions.

1.6 Using this manual

This document is intended for system programmers who are targeting their software for operation on an HVM Monitor or Hypervisor. It is also intended as a reference for HVM implementors.

1.7 Terminology

Except where otherwise noted, the terms used to describe HVM are identical to those used to describe the Hexagon processor architecture.

Table 1-1 Terminology

Term	Definition
Event record	Data structure passed on the Guest-mode stack during a virtual event, providing context information necessary for the handling of the event.
Guest mode	Operating mode under which operating systems run on HVM to provide various services to user applications.
GBADVA	Guest Bad Virtual Address
GELR	Guest Event Link Register
GOSP	Guest Other Stack Pointer
GSR	Guest Status Register
HVM	Hexagon Virtual Machine Virtualized programming environment in which a client software system runs. It includes one or more virtual processors. logical memory, and a virtual interrupt controller.
Logical memory	Virtual machine memory accessible by HVM clients. Logical memory is the memory that is not mapped by the HVM memory management unit (MMU) for use in constructing virtual memory maps in HVM.
PTE	Page table entry
User mode	Operating mode under which most HVM applications run.
Virtual event	Asynchronous invocation of Guest mode by the virtual machine.
Virtual instruction	An operation requesting a function of HVM that cannot be expressed by a native Hexagon processor instruction. Virtual instructions are generally implemented as Hexagon instructions which trap to the underlying hardware Supervisor mode, which performs the necessary function.
Virtual machine	HVM instance running on the VMM.

Table 1-1 Terminology (continued)

Term	Definition
Virtual processor	HVM resource that is necessary to run a program. A virtual machine can support multiple instances of a virtual processor, with each instance executing concurrently and sharing common memory and I/O resources.
VMM	Virtual Machine Monitor Software layer which underlies HVM. It manages the hardware resources and provides them to HVM in their virtualized form.

1.8 Notation

This document uses the same notational conventions used in the *Hexagon Programmer's Reference Manual*. For details see the appropriate version of that document.

2 HVM User Mode

2.1 Overview

Except for the restrictions noted below, HVM User mode is identical to the processor architecture defined in the *Hexagon Programmer's Reference Manual*.

Most HVM clients run in User mode.

2.2 Restrictions

HVM User mode has the following restrictions:

- Memory addressing

2.2.1 Memory addressing

The VMM reserves the high 16 MB of the User-mode address space (addresses `0xff00_0000` to `0xffff_ffff`) for its own use.

Any attempts by an HVM client to load, store, or execute from addresses in this range will result in a protection-violation event being posted to Guest mode.

NOTE This restriction also applies to the Guest-mode address space (Section 3.3.1).

3 HVM Guest Mode

3.1 Overview

HVM Guest mode is a superset of HVM User mode (Chapter 2).

In addition to the standard Hexagon processor architecture supported by User mode, Guest mode defines *virtual instructions* and *virtual events* to support the implementation of protected, multi-user, multi-tasking operating systems that run on HVM.

Virtual instructions are extensions to the Hexagon instruction set – they invoke HVM operations that cannot be performed by a regular Hexagon instruction. Virtual instructions are generally implemented as Hexagon traps which switch the processor to its Supervisor mode.

Virtual events are asynchronous transfers from User to Guest mode – they provide a controlled way to switch between the HVM modes. Virtual events can be triggered by exceptions or interrupts.

NOTE HVM virtual instructions execute like regular processor instructions, but are always treated as solo instructions which cannot be grouped in packets with other instructions.

Appendix A provides details on the encoding and semantics of the virtual instructions.

3.2 Guest mode capabilities

HVM Guest mode has the following capabilities:

- Event handling
- Interrupts
- Exceptions
- Cache control
- Memory management
- Processor resource management

3.2.1 Event handling

HVM can transfer control asynchronously to handle events:

- Handler vectors must be registered using privileged virtual instruction `vmsetvec`
- Event handling is terminated with virtual instruction `vmrte`

For more information see Chapter 5.

3.2.2 Interrupts

HVM supports Hexagon interrupts as a special case of asynchronous events:

- Interrupts are individually maskable
- Interrupts can be collectively enabled and disabled
- Interrupts are managed with virtual instructions `vmintop`, `vmsetie`, and `vmgetie`

For more information see Chapter 6.

3.2.3 Exceptions

HVM supports Hexagon exceptions as another type of event:

- Exceptions are not maskable (unlike interrupts)

For more information see Chapter 7.

3.2.4 Cache control

HVM supports certain Hexagon cache control operations:

- Privileged operations (Guest mode only)
- Managed with virtual instruction `vmcache`

For more information see Chapter 8.

3.2.5 Memory management

HVM implements a virtual memory management unit:

- Privileged operations (Guest mode only)
- Autonomous page table walking
- Page tables in canonical HVM format
- Managed with virtual instructions `vmnewmap` and `vmclrmap`

For more information see Chapter 9.

3.2.6 Processor resource management

HVM supports virtual processors and a fine-grained timer:

- Privileged operations (Guest mode only)
- Create and destroy concurrently-executing virtual processor instances
- Virtual processors managed with virtual instructions `vmstart` and `vmstop`
- High-precision 64-bit timer
- Timer managed with virtual instructions `vmgettime` and `vmsettime`

For more information see Chapter 10.

3.3 Restrictions

HVM Guest mode has the following restrictions:

- Memory addressing

3.3.1 Memory addressing

The VMM reserves the high 16 MB of the Guest-mode address space (addresses `0xff00_0000` to `0xffff_ffff`) for its own use.

Any attempts by an HVM client to load, store, or execute from addresses in this range will result in a protection-violation event being posted to Guest mode.

NOTE This restriction also applies to the User-mode address space (Section 2.2.1).

4 Initial State

4.1 Overview

This chapter describes the initial state of the following HVM resources:

- Virtual processor
- Registers
- Memory

4.2 Virtual processor

A virtual processor begins executing in Guest mode. The virtual MMU is always active.

4.3 Registers

For virtual processors the initial value of the program counter (PC) and stack pointer are specified externally: either as parameters to commands passed to the VMM, or as arguments to the virtual instruction `vmstart` which creates a new virtual processor instance (Section 10.4).

The initial values of all other registers are undefined – these registers must be initialized by software starting at the initial PC.

Implementors Note: *While register values are undefined at the startup of a virtual processor, if the hardware processor resources are re-assigned, the register values of trusted virtual processors must never be inherited by untrusted virtual processors.*

4.4 Memory

When an initial program is loaded by the VMM, the program image may contain initialized data. Otherwise the initial memory state of HVM consists of undefined memory values.

Implementors Note: *While memory values are undefined at startup of a virtual machine, in the event that the hardware resources are re-assigned, memory values of trusted virtual machines must never be inherited by untrusted virtual machines.*

4.5 Initial memory map

In the HVM initial memory map, the logical memory segment that the initial program is loaded into is mapped to the identical virtual address range, with consecutive virtual pages mapped 1:1 to consecutive logical pages.

The page tables describing the initial map are outside the initial logical address space, and cannot be modified by HVM clients (Section 2.2.1, Section 3.3.1).

Any other mapping must be created by Guest-mode software constructing a new set of page tables and activating them with virtual instruction `vmnewmap` (Section 9.5).

5 Event Model

5.1 Overview

HVM supports *virtual events* (or more simply, *events*) as a means for transferring control between User-mode applications and Guest-mode operating systems. HVM events can be caused by Hexagon processor exceptions or interrupts.

5.2 Event model

Events are specified by their assigned event number. When an event occurs, HVM uses the event number as an index into the event vector table to determine which event handler will process the event. The vector table and handlers are defined by the guest operating system.

Before it begins receiving events, the guest operating system must register the Guest Event Vector Base (GEVB) value with HVM. It does this with the virtual instruction vmsetvec.

HVM responds to an event by storing event-related information (called the *event record*) in the dedicated event registers (Section 5.3), and then transferring control to the event vector specified by the event number.

Event handlers access the event registers with the virtual instructions vmgetregs and vmsetregs – the guest operating system is responsible for managing these registers.

When an event is dispatched to a handler, interrupts are disabled to the virtual processor. Interrupts are re-enabled either explicitly when the handler executes the virtual instruction vmsetie, or implicitly when the handler returns by executing the virtual instruction vmrte. (vmrte restores the interrupt enable value from the event record.)

When HVM transfers control between User and Guest modes, it swaps the contents of the Guest Other Stack Pointer (GOSP) and R29 (the Hexagon processor stack pointer).

If an event handler causes an additional event, the original event record is lost because the event registers are overwritten. Similarly, when returning from an event, the event registers are reloaded by the guest operating system before executing the virtual instruction vmrte. Care must be taken to ensure that no event occurs during this time.

NOTE If an event occurs before GEVB is configured, HVM terminates the current virtual processor (Section 10.4).

5.3 Event registers

Table 5-1 defines the four 32-bit registers that are used to store HVM event records (Section 5.2).

Table 5-1 Event registers

Register	Alias	Name
GELR	G0	Guest Event Link Register
GSR	G1	Guest Status Register
GOSP	G2	Guest Other Stack Pointer
GBADVA	G3	Guest Bad Virtual Address value

The Hexagon processor instruction set does not provide instructions for directly accessing the event registers. Instead, these registers are accessed with the virtual instructions `vmgetregs` and `vmsetregs`.

NOTE Future Hexagon processor versions may support direct access to the event registers, eliminating the need for these virtual instructions.

5.3.1 GELR

`GELR` is the Guest Event Link Register.

It contains a 32-bit virtual address value specifying where the virtual processor resumes executing after an event is serviced.

If a guest operating system wishes a processed event to return to a different address in User mode (for example, in the case of a user-level context switch), the `GELR` value can be modified before the virtual instruction `vmrte` is executed.

5.3.2 GSR

GSR is the Guest Status Register.

It provides information about the virtual processor state when the event occurred, and about the nature of the event beyond what is implicitly indicated by the event vector.

31	30	29				24	23					16	15													0
UM	IE	Reserved					Reserved						Cause													

Table 5-2 describes the fields of the GSR word.

Table 5-2 GSR field encodings

Bits	Name	Description
31	UM	Set if virtual processor was executing in User mode when the event was taken, clear otherwise.
30	IE	Set if virtual processor had interrupts enabled when the event was taken, clear otherwise.
15:0	Cause	Cause code associated with an Exception or Interrupt event.

NOTE The User Mode and Interrupt Enable mask of the virtual processor are restored from GSR by a vmrte instruction.

5.3.3 GOSP

GOSP is the Guest Other Stack Pointer.

When executing in User mode, this register contains the Guest-mode stack pointer.

If an event occurs during User-mode execution, HVM swaps the contents of GOSP and R29 (the Hexagon processor stack pointer). Thus when the event vector is reached, R29 contains the Guest-mode stack pointer, and GOSP stores the User-mode stack pointer.

When the virtual instruction vmrte switches from Guest back to User mode, HVM swaps R29 and GOSP again, saving the Guest-mode stack pointer in GOSP and restoring the User-mode stack pointer in R29.

5.3.4 GBADVA

GBADVA is the Guest Bad Virtual Address value.

When servicing a memory-related event (page fault, alignment error, permissions violation), GBADVA contains the 32-bit virtual address that caused the event.

5.4 Event types

Table 5-3 summarizes the event types.

Table 5-3 Event types

Event Number	Event Name	Event Description	Notes
0	Reserved		
1	Machine Check	Unrecoverable virtual machine state (Section 7.5)	Non-maskable exception. Virtual machine execution terminates if not handled.
2	General Exception	Precise internal hardware or software exception (Section 7.3)	Non-maskable exception
3, 4	Reserved		
5	Trap0	SW Trap0 instruction (Section 7.4)	Non-maskable exception
6	Reserved		
7	Interrupt	General external interrupts (Chapter 6)	Maskable. Interrupt priority decreases with increasing interrupt number, Interrupt0 being highest.

NOTE The Hexagon architecture specifies that if an instruction packet contains multiple exception-causing instructions, then Slot 1 exceptions are resolved before Slot 0 exceptions. If a single instruction has multiple exceptions, the priority is listed in the table from highest (first row) to lowest (last row).

5.5 Event vector registration

Guest-mode software can register a service vector for an event by executing the virtual machine instruction `vmsetvec`, which is passed a 32-bit Guest-mode virtual address.

Events begin executing at the address computed by the following formula:

```
GEBV + (event_number * 4)
```

5.6 Event handler return

Event handlers return by executing the virtual instruction `vmrte`. This instruction uses the values stored in the event registers to restore the following states:

- User/Guest mode and interrupt enable (stored in `GSR.UM` and `GSR.IE`)
- Program counter (stored in `GELR`)
- Stack pointer (stored in `GOSP`) if the value in `GSR.UM` indicates that the mode switch is from Guest to User

NOTE `vmrte` assumes that a valid event record is stored in the event registers.

5.7 Virtual instructions for event management

Table 5-4 summarizes the virtual instructions used for event management.

Table 5-4 Virtual instructions for event management

Instruction	Description
`vmsetvec`	Register event vector table
`vmgetregs`	Get event register values
`vmsetregs`	Set event register values
`vmsetie`	Enable interrupts
`vmrte`	Return from event servicing

NOTE For details on these instructions see Appendix A.

5.8 Programmers note

An HVM instance begins executing in Guest mode, so the only way to execute software in User mode is to perform the following steps:

1. Set the event registers so the `GSR.UM` bit is set, and the `GELR` value specifies code to be executed in User mode.
2. Execute a `vmrte` instruction.

6 Interrupts

6.1 Overview

In HVM interrupts are handled as a special case of events. When an interrupt event is dispatched, the `GSR.Cause` value in the event record is set to the interrupt number.

Interrupts can be enabled or completely disabled per virtual processor by setting the virtual state `Interrupt Enabled`. When an event is taken, `Interrupt Enabled` is cleared and the previous `Interrupt Enabled` state is stored in `GSR`.

6.2 Virtual interrupt controller

HVM defines a *virtual interrupt controller* to manage interrupt events flowing into the multiple virtual processors.

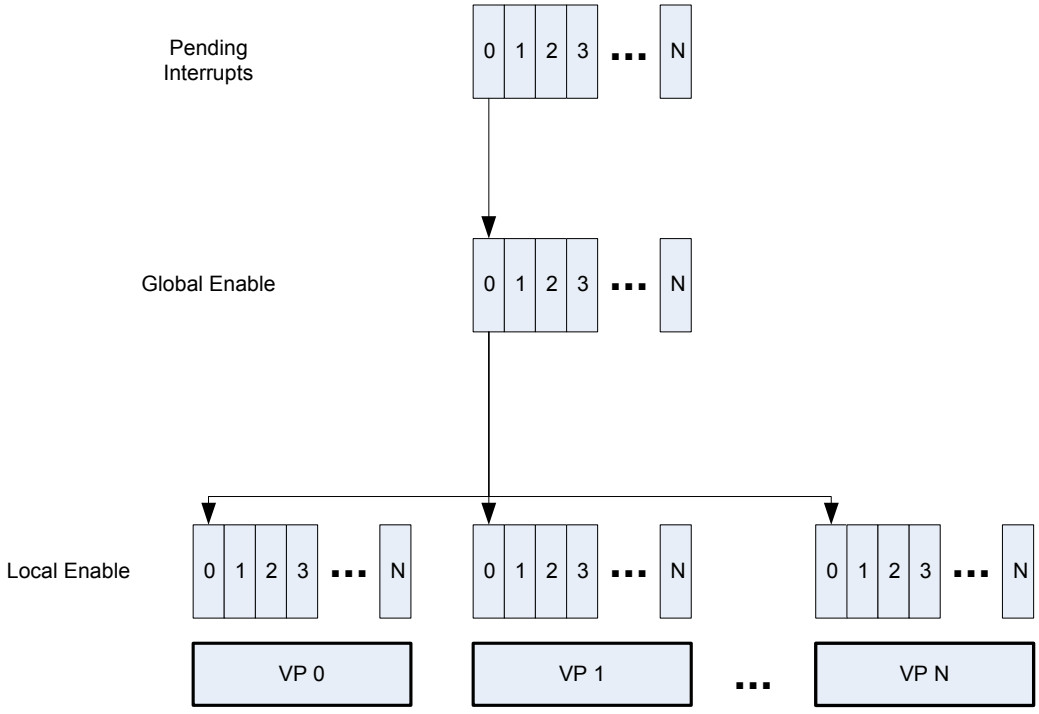

Figure 6-1 Virtual interrupt controller

Each interrupt has a global enable, as well as a per-virtual processor local enable:

- If an interrupt is disabled globally, no virtual processor takes the interrupt.
- If an interrupt is disabled locally, the specific virtual processor does not take the interrupt.

By manipulating the local and global enable values, interrupts can be steered to specific virtual processors, or received by any available virtual processor.

When an interrupt arrives, its bit is set in the pending-interrupt bit vector. If the interrupt is globally and locally enabled for a virtual processor that has interrupts enabled, the virtual processor may take the interrupt: the PC changes to the interrupt event, and the interrupt number is stored in `GSR.CAUSE`. In addition, the interrupt is cleared from the pending register and automatically disabled globally.

NOTE At most one virtual processor takes a given interrupt. Interrupting a virtual machine and clearing the pending bit and global enable are atomic operations.

6.3 Interrupt type and polarity

The VMM is responsible for configuring interrupt type and polarity.

6.4 Interrupt masks

HVM has both a per-interrupt global enable bit and a per-interrupt, per-virtual-processor enable bit. These bits can be modified by the virtual instruction `vmintop`, and can be used to mask interrupts either for a specific processor or for all processors.

6.5 Interrupt acknowledgement

When HVM posts an interrupt event, the interrupt is disabled globally. Thus, any further occurrences of that interrupt are not posted on any virtual processor until the interrupt service routine re-enables it using the `GLOBEN` operation of the virtual instruction `vmintop`.

6.6 Interrupt enable

Each virtual processor has an `Interrupt Enabled` status. This status can be checked using the virtual instruction `vmgetie`, and modified using the virtual instruction `vmsetie` (as well as `vmrte`). When a virtual processor has the `Interrupt Enabled` status disabled (set to zero), it takes no interrupts.

The `Interrupt Enabled` status is automatically set to `Disabled` on every taken event (exception or interrupt), as well as when the virtual processor starts.

6.7 Virtual instructions for interrupt management

Table 6-1 summarizes the virtual instructions used for interrupt management.

Table 6-1 Virtual instructions for interrupt management

Instruction	Description
`vmintop`	Perform interrupt-related operation: ■ Globally enable interrupt ■ Globally disable interrupt ■ Locally enable interrupt ■ Locally disable interrupt ■ Set interrupt affinity ■ Post software interrupt ■ Check for pending interrupt ■ Take pending interrupt ■ Clear pending interrupt ■ Get interrupt pending, global enable, local enable
`vmgetie`	Return current `Interrupt Enabled` state
`vmsetie`	Set `Interrupt Enabled` state and return old state

The `vmintop` operation “Set interrupt affinity” locally enables an interrupt for the specified virtual processor, and locally disables it for all other virtual processors.

NOTE For details on these instructions see Appendix A.

7 Exceptions

7.1 Overview

Exceptions are non-interrupt events, and are not maskable like interrupts.

Exceptions can be precise or imprecise. When a precise exception event is handled, `GELR` contains the following value:

- General exceptions – The address of the instruction packet that caused the exception
- Trap exceptions – The packet following the `trap` instruction

NOTE When an exception occurs as a consequence of a load or store address, `GBADVA` contains the address associated with the exceptional operation.

7.2 Exception classes

HVM supports three classes of exceptions, each of which is handled separately:

- General exceptions
- Trap exceptions
- Virtual machine check

7.3 General exceptions

General exceptions in HVM correspond closely to precise exceptions in the underlying Hexagon processor. The general exception value is stored in the `GSR.Cause` field of the event record.

Table 7-1 General exceptions

Exception	Description	GBADVA
0x00	Reserved	Undefined
0x01	Precise Bus or Memory Error.	Undefined
0x02-0x10	Reserved	Undefined
0x11	Execute Protection Violation. Attempt to fetch instructions from virtual memory page without execute permission.	Undefined
0x12-0x13	Reserved	Undefined
0x14	User Access Violation. Attempt to fetch instructions in User mode from virtual memory page without user access permission.	Undefined
0x15	Invalid Instruction Packet. Attempt to execute malformed instruction packet, or one containing reserved or invalid opcodes or combinations of operations.	Undefined
0x16-1A	Reserved	Undefined
0x1B	Privilege Violation. Attempt to execute privileged instruction or virtual instruction in User mode.	Undefined
0-x1C	Misaligned Program Counter. Attempt to transfer control to misaligned instruction packet address.	Undefined
0x1D-1F	Reserved	Undefined
0x20	Load to Misaligned Address	Misaligned Load Address
0x21	Store to Misaligned Address	Misaligned Store Address
0x22	Load Protection Violation. Attempt to load data from virtual memory page without read permission.	Protected Load Address
0x23	Store Protection Violation. Attempt to store data to virtual memory page without write permission.	Protected Store Address
0x24	Load User Access Violation. Attempt to load data in User mode from page without user access permission.	Protected Load Address
0x25	Store User Access Violation. Attempt to store data in User mode to page without user access permission.	Protected Store Address
0x26-27	Reserved	Undefined
0x28	Cache Conflict. Instruction packet contains a set of instructions and address modes that are incompatible with cache attributes of referenced addresses.	Undefined
0x29	Instruction packet with destination register collision	Undefined
0x30-0xFF	Reserved	Undefined

7.4 Trap exceptions

Trap exceptions are triggered when a *trap0* instruction is executed by an HVM client.

The 8-bit value encoded in the *trap0* instruction is stored in the `GSR.Cause` field of the event record.

NOTE On trap exceptions `GELR` points to the packet following the trap instruction, and not to the trap instruction itself.

7.5 Virtual machine check

Machine Check exceptions indicate the occurrence of potentially unrecoverable failures in the virtual machine or its underlying hardware.

HVM software can register and handle machine check exceptions to perform emergency shutdown operations or to attempt recovery. Normal termination of a machine check event handler is performed with a `vmstop` virtual instruction (and not `vmrte`).

If the Guest-mode software has fully handled the failure (for example, the fault has been isolated to a single User-mode program which can be terminated in isolation), then it can terminate with a `vmrte`. However, it is implementation-dependent whether the virtual machine monitor will allow resumption of pre-event processing.

Table 7-2 Machine check exceptions

Exception	Description
0x00	Shutdown. Administrative shutdown of virtual machine
0x01	Invalid Guest mode stack pointer
0x02	Reserved
0x03	Invalid logical page mapping in page table detected at runtime
0x04-0x1F	Reserved
0x20	Imprecise Data Abort to underlying hardware
0x21-0x2F	Reserved
0x30	NMI to underlying hardware
0x31-0xFFFF	Reserved

8 Cache Control

8.1 Overview

The user-level Hexagon cache management instructions (`dcfetch`, `icinva`, `dccleaninva`, `dccleana`, `dcinva`) can be used in HVM clients.

Additional cache control operations that must be handled by the VMM are performed with the virtual instruction `vmcache`.

8.2 HVM cache operations

The virtual instruction `vmcache` is used to perform various cache operations that are difficult (or inefficient) to perform with the user-level Hexagon cache control instructions (`dcfetch`, `icinva`, `dccleaninva`, `dccleana`, `dcinva`).

`vmcache` additionally enables `cleaninva`-like operations over large virtual address areas to be optimized by the VMM (Section A.2.1).

NOTE `vmcache` operations are non-destructive of data in the cache.

Table 8-1 lists the HVM-specific cache control operations.

Table 8-1 HVM cache control operations

Operation	Description
ICKILL	Non-destructive invalidate of entire instruction cache
DCKILL	Non-destructive invalidate of entire data cache
L2KILL	Non-destructive invalidate of entire L2 data cache
DCCLEANINVA	Non-destructive invalidate of virtual address range, starting at address in R1 for the number of bytes in R2, through full D-cache hierarchy.
ICINVA	Non-destructive invalidate of virtual address range, starting at address in R1 for the number of bytes in R2, through full I-cache hierarchy.
IDSYNC	Ensure that Instruction cache can observe all data in specified address range. NOTE - Similar to DCCLEANINVA followed by ICINVA.
Reserved	Reserved (No-op)

8.3 Virtual instructions for cache control

Table 8-2 summarizes the virtual instructions used for HVM cache control.

Table 8-2 Virtual instructions for event management

Instruction	Description
`vmcache`	Perform one of several HVM cache control operations

For details on these instructions see Appendix A.

9 Memory Management

9.1 Overview

HVM maps virtual addresses `0x0000_0000` to `0xfeff_ffff` onto a 32-bit logical address space, using either a list of translations, or a 1- or 2-level virtual page table scheme. The translations are visible to and writable by a guest operating system running on HVM.

The VMM is responsible for validating and transforming the information as necessary when filling the physical processor TLB. In normal operation, translations are loaded on-demand by the VMM.

9.2 Underlying logical/physical memory

“Logical” addresses are addresses not translated by the HVM MMU. Depending on the virtual machine implementation and configuration, they may be hardware physical addresses or addresses mapped by the VMM.

NOTE The ranges of logical memory accessible in HVM may be determined by platform-specific operations. Attempts to map and access logical addresses which are not accessible to a virtual machine will result in a protection violation exception event.

9.3 Linear translations

Virtual-to-logical address translations can be specified by a list of translations. Figure 9-1 shows the format of a list entry. Table 9-1 describes the fields in a list entry.

The VMM traverses the translation list in a linear fashion. The list is terminated by a translation entry which has all bits set to zero.

NOTE This format requires less space to hold simple memory layouts, and may be preferable for more highly-embedded Guest-mode environments.

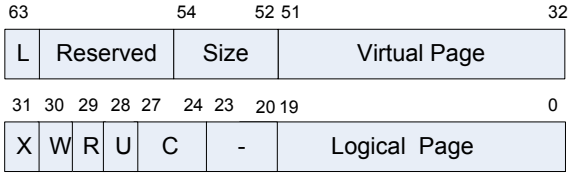

Figure 9-1 Translation list entry

Table 9-1 Translation list entry fields

Field	Description
Logical Page	Logical Page Number that the corresponding Virtual Page maps to
Virtual Page	Virtual Page Number that is matched against the load or store address
Size	Page size: ■ 000: 4 KB ■ 001: 16 KB ■ 010: 64 KB ■ 011: 256 KB ■ 100: 1 MB ■ 101: 4 MB ■ 110: 16 MB NOTE - All other values are reserved. Programming an entry with a reserved pattern results in undefined behavior.
U, R,W, X	Permissions: ■ U: User-enable. Set if user accesses are allowed for this page ■ R: read-enable. Set if reads are allowed from this page. ■ W: write-enable. Set if writes are allowed to this page. ■ X: execute-enable. Set if execution is allowed for this page. If permissions are violated, a precise exception will be raised with the appropriate cause code (load, store, or fetch).
C	Cacheability (Section 9.4.3)
L	Link bit. If this bit is set, the current entry points to next set of translation entries. Bits 31:0 specify address of the next translation in the list; the other bits in the entry are ignored.
Reserved	Reserved field. Field currently ignored by HVM. Because it may be used in future HVM versions, the recommended value for this field is 0.

9.4 Virtual page table entries

HVM defines two levels of virtual page table. The first level breaks down the virtual address space into 1020 4MB segments, with each segment represented by a first-level page table entry (PTE).

A first-level PTE always contains the size of the virtual memory page that is mapped:

- In the case of pages 4MB or greater, the first-level entry contains the translation and permissions information for the page.
- For pages less than 4MB, the first-level entry contains a pointer to an array of second-level entries.

A first-level PTE encodes the entry type and page size in its 3-bit S field (Figure 9-3). The definition of the remaining bits varies according to the entry type.

Figure 9-2 Page table entry (generic)

Table 9-2 lists the entry types for an L1 page table entry.

Table 9-2 L1 page table entry types

S Field Value	Entry Type	Page Size	L2 Entries	Address Bits
000	Page Directory Entry	4KB	1024	31:12
001	Page Directory Entry	16KB	256	31:10
010	Page Directory Entry	64KB	64	31:8
011	Page Directory Entry	256KB	16	31:6
100	Page Directory Entry	1MB	4	31:4
101	Page Table Entry	4MB	N/A	N/A
110	Page Table Entry	16MB	N/A	N/A
111	Invalid	Invalid	N/A	N/A

If the size specified in S is greater than or equal to 4M, a second level of page table is not necessary to define the mapping: the entry defines the page mapping and permissions directly. For smaller pages, the entry is interpreted as a directory entry which points to a second-level table.

NOTE Referencing a virtual address which resolves to a PTE marked as invalid (111) results in a protection violation exception event.

Because the granularity of the first-level page table is 4MB, the PTE for a 16MB virtual page must be replicated for each of the 4 entries in the first-level page table that correspond to the 16MB virtual address range.

9.4.1 Page directory entries

A PTE contains (in addition to the S field) a pointer to a second-level page table. The most significant bits of the entry are used as the most significant bits of the logical address of the L2 table.

Figure 9-3 shows the format of a page directory entry.

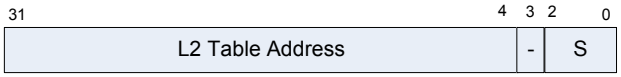

Figure 9-3 Page directory entry

NOTE L2 tables must be aligned to the size of the table.

9.4.2 Page table entries

PTEs are used to define the virtual-to-logical translation for a specific memory page.

Instead of a pointer to a second-level page table, a PTE can specify a 4MB or 16MB translation at the L1 level. Figure 9-4 and Figure 9-5 show the corresponding entry formats (with the S field set to specific values per Table 9-2).

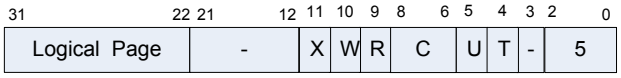

Figure 9-4 Page table entry (L1 – 4MB)

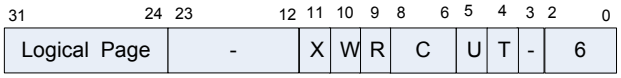

Figure 9-5 Page table entry (L1 – 16MB)

Second-level page tables specify translations for a contiguous 4MB of the address space. Each entry has the same translation size. This uniform size is specified at the first level. Figure 9-6 shows a second-level PTE.

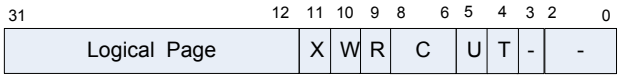

Figure 9-6 Page table entry (L2)

Table 9-3 describes the fields in a PTE.

Table 9-3 Page table entry fields

Bits	Mnemonic	Description
31:12	LPN	Logical Page Number. High-order bits of page-aligned logical address corresponding to mapped page. The number of bits actually used depends on specified page size. See Table 9-4.
11	X	Execute Permission. HVM has execute access to page. Attempts fetch instructions from an address within page causes an Execute Permission Exception event if the X bit is cleared (0).
10	W	Write Permission HVM has write access to page. Attempts to store data to an address within the page causes a Write Permission Exception event if W bit is cleared (0).
9	R	Read Permission HVM has read access to page. Attempts to load data from an address within the page causes a Read Permission Exception event if the R bit is cleared (0).
8:6	C	Cache Attributes See Section 9.4.3.
5	U	User Mode If the bit is set (1), User mode programs may reference the page. If it is cleared (0), read, write, or execute references in User mode result in a User Access Exception event.
4	T	Trusted Behavior depends on the system trust model. Recommended setting is cleared (0).
Varies	–	Unused. Ignored by HVM.

NOTE For L1 entries, the size field must be set correctly for determining page size. For L2 entries, the size field is ignored and may be used by software.

If none of the R, W, or X bits are set, the virtual map is considered invalid, and any access (load, store, fetch) results in a Page Fault Exception event.

Table 9-4 LPN bits used as function of page size

Page SIze	LPN Bits Used
4KB	31:12
16KB	31:14
64KB	31:16
256KB	31:18
1MB	31:20
4MB	31:22
16MB	31:24

NOTE HVM ignores any LPN bits that are not required to address an aligned page of the specified size.

9.4.3 Cache attributes

In a page table entry (Section 9.4.2) the C field controls the cache attributes of the virtual page. The cache behavior is undefined if virtual aliases map the same logical page with different cache attributes.

C is relevant only for “terminal” PTEs which resolve to a target logical page address.

Table 9-5 Cache attribute types

C Field Value	L1 Cache Policy	L2 Cache Policy
0	Write-back	Non-cacheable
1	Write-through	Non-cacheable
2	Reserved	
3	Reserved	
4	Device Access	
5	Write-through	Cacheable
6	Uncached Memory	
7	Write-back	Cacheable
8-15	Reserved	

The cache attribute `Device Access` should be used whenever loads or stores have side effects.

The cache attribute `Uncached Memory` does not allow memory to be cached. However, the implementation is allowed to replay loads and stores.

Implementors Note: *The HVM implementation is free to replay loads and stores for pages assigned* `Uncached Memory`*. (The Hexagon processor will in fact do this.)*

NOTE To maintain compatibility with future HTM versions, the reserved cache attribute settings should not be used.

HVM may choose to reduce the cacheability of a translation for reasons of correctness or Quality-of-Service.

9.5 Setting new memory maps

The virtual instruction `vmnewmap` establishes a new logical-to-virtual memory map for HVM.

`vmnewmap` accepts as arguments the logical addresses of either a segment table (Section 9.4) or linear translation list (Section 9.3). The instruction immediately following `vmnewmap` executes with the new memory map in effect.

Implementors Note: *If an invalid page table is passed to* `vmnewmap`*, the HVM implementation is free to determine whether* `vmnewmap` *returns with a value indicating failure, or whether a subsequent attempt to de-reference an address with an invalid mapping causes a virtual machine check. For more information see Section 7.5.*

9.6 Flushing stale memory maps

The virtual instruction `vmclrmap` flushes a possibly-stale virtual mapping.

`vmclrmap` accepts as arguments a range of virtual addresses. It causes copies of virtual address translations to be purged from the virtual machine (including the TLB) for the specified range of virtual addresses.

Subsequent accesses to the memory range use the values in the memory map tables.

9.7 Virtual instructions for memory management

Table 9-6 summarizes the virtual instructions used for memory management.

Table 9-6 Virtual instructions

Instruction	Description
`vmnewmap`	Register new logical-to-virtual memory map.
`vmclrmap`	Flush a possibly-stale virtual mapping.

For details on these instructions see Appendix A.

10 Processor Resource Management

10.1 Overview

HVM supports the following facilities for processor resource management:

- Fine-grained system timer
- Ability to suspend processor execution
- Ability to create new virtual processors

10.2 Timer

HVM supports a fine-grained timer which is accessed through the virtual instructions `vmgettime` and `vmsettime`:

- `vmgettime` returns a 64-bit value representing the current time.
- `vmsettime` resets the timer to the 64-bit value passed as an argument.

Implementors Note: *The HVM implementation is free to define whether the timer corresponds to CPU time or wall-clock time. For example, if a Virtual Machine Monitor schedules out a virtual CPU, the timer may or may not continue to increment while the virtual CPU is not executing.*

10.3 Processor suspension

Guest-mode software on HVM can suspend the execution of a virtual processor in two ways:

- The virtual instruction `vmwait`
- The virtual instruction `vmyield`

`vmwait` suspends execution pending an interrupt event. After the next event is handled, the virtual instruction `vmrte` that terminates the event servicing will result in execution resuming at the instruction after the `vmwait` instruction.

`vmyield` suspends execution temporarily, independently of any virtual processor events. It auses the VMM to schedule other virtual processors at the same or higher priority, without the need for pre-emption.

NOTE It is recommended that the guest operating system yield while attempting to acquire a spinlock. This ensures that a virtual processor holding the spinlock has an opportunity to execute.

10.4 Processor creation

Guest-mode software can create new instances of virtual processors with the virtual instruction `vmstart`. The newly-created instances execute concurrently with the existing processor instances.

A new virtual processor instance begins execution in Guest mode, fetching instructions from the address specified as an argument to `vmstart`. The new instance begins execution with the same memory map as the virtual processor instance that executed the `vmstart`.

Each virtual processor has a unique 32-bit identifier, which can have values in the following range:

```
0 .. (max_number_of_supported_virtual_processors - 1)
```

A virtual processor ID value can never have the value -1 (all ones). The initial virtual processor instance has a virtual processor ID of zero. Software running on a virtual processor can access its virtual processor ID by executing the virtual instruction `vmvpid`.

The inverse of `vmstart` is the virtual instruction `vmstop`. When a virtual processor executes `vmstop`, the following events occur:

- Execution of the virtual processor is terminated.
- The instruction following the `vmstop` is not issued.
- The resources associated with the virtual processor are freed and made available for use by any new virtual processor instance created by a subsequent `vmstart` operation.

10.5 Virtual instructions for processor management

Table 10-1 summarizes the virtual instructions used for processor resource management.

Table 10-1 Virtual instructions for processor management

Instruction	Description
`vmgettime`	Get the 64-bit virtual processor timestamp
`vmsettime`	Set virtual processor timestamp value.
`vmwait`	Wait on an interrupt event.
`vmyield`	Voluntarily reschedule virtual processor resources.
`vmstart`	Start a new, concurrently executing virtual processor.
`vmstop`	Terminate virtual processor execution.
`vmvpid`	Get virtual processor ID.

For details on these instructions see Appendix A.

11 Trust Model

11.1 Overview

The HVM environment provides a trust model which is distinct from, but interoperable with the trust model of other cores in the system.

12 Debug

12.1 Overview

HVM debug support must reconcile two factors:

- The development requirement for simple and effective debugging of programs executing on the virtual machine.
- The operational requirement for security and isolation.

12.2 User-mode software debug

User-mode software can be debugged using Guest-mode facilities. For example:

- Generating page faults on memory references
- Inserting `trap0` instructions for breakpoints and single-step operation

These facilities are sufficient to implement a UNIX-style `ptrace` capability for the guest operating system.

12.3 Guest-mode software debug

The VMM can use the Hexagon processor's supervisor-mode facilities to debug User-mode code. These same process-level facilities can be used to debug Guest-mode software.

Implementors Note: *The HVM implementation is free to determine whether the VMM accepts debugging commands directly from some operator communications channel (such as a serial port or a JTAG interface), or whether the VMM provides an API to trusted-debug-agent module which translates from operator/host debugger commands on the communications channel and VMM debug API functions.*

Under no circumstances, however, can an untrusted VMM or operator debug agent generate trusted load or store cycles, either directly or indirectly, through a trusted system that is being debugged.

A HVM Instructions

A.1 Overview

HVM provides virtual instructions and events which control the transitions between User and Guest modes, and which support the implementation of protected, multi-user, multi-tasking operating systems that run on HVM.

Virtual instructions are extensions to the Hexagon instruction set – they invoke HVM operations that cannot be performed by a regular Hexagon instruction.

This appendix provides detailed descriptions of the HVM virtual instructions. The instructions are listed alphabetically.

Table A-1 summarizes the virtual instructions.

Table A-1 HVM virtual instruction summary

Mnemonic	Function	Encoding	Inputs	Outputs
vmversion	Request Virtual Machine Version	trap1(#0)	R0 = Requested VM Version	R0 = Configured VM Version
vmrte	Return from Event	trap1(#1)	Event Record in g3-g0	N/A
vmsetvec	Set event vector	trap1(#2)	R0 = Vector Table Address	R0 = 0 on success, otherwise -1
vmsetie	Set interrupt enable state	trap1(#3)	R0 1 to enable, 0 to disable	R0 = Previous enable state in bit 0
vmgetie	Get interrupt enable state	trap1(#4)	N/A	R0 = Enable state in bit 0
vmintop	Interrupt Operation	trap1(#5)	R0 = Interrupt Op R1-R4: Depends on Op	R0 depends on Op.
vmclrmap	Clear virtual memory map	trap1(#10)	R0 = Starting VA R1 = Length in bytes	R0 = 0 on success, otherwise -1
vmnewmap	Set new virtual memory map	trap1(#11)	R0 =Logical address of new segment table R1 =Type of translations	R0 = 0 on success, otherwise negative error code
vmcache	Virtual Machine cache control	trap1(#13)	R0 = Operation to be performed R1 = Starting virtual address R2 = Length in bytes	R0 = 0 on success, otherwise -1
vmgettime	Get Virtual Machine Time	trap1(#14)	N/A	R0 = least significant 32 bits of timestamp R1 = most significant 32 bits of timestamp
vmsettime	Set Virtual Machine time	trap1(#15)	R0 = least significant 32 bits of timestamp R1 = most significant 32 bits of timestamp	N/A
vmwait	Wait for next interrupt	trap1(#16)	N/A	R0 = Interrupt number of re-activating event
vmyield	Voluntarily reschedule virtual processor	trap1(#17)	N/A	N/A
vmstart	Create new virtual processor instance	trap1(#18)	R0 = Starting execution address R1 = Starting stack pointer	R0 = Virtual processor number of new virtual processor on success, otherwise -1
vmstop	Terminate current virtual processor instance	trap1(#19)	N/A	N/A
vmvpid	Obtain virtual processor ID	trap1(#20)	N/A	R0 = Virtual processor number of virtual processor executing the instruction
vmsetregs	Set Guest Registers	trap1(#21)	R0-3 hold G0-3 values	N/A
vmgetregs	Read Guest Registers	trap1(#22)	N/A	R0-3 hold G0-3 values.

A.2 Instruction properties

HVM virtual instructions have the following properties:

- They are privileged, and can be executed only in Guest-mode software. If executed in User mode, they will generate a privilege violation exception.
- By convention they take their inputs (if any) from the Hexagon processor registers R0–R4, and return their outputs in registers R0 and R1, as necessary.
- They are solo instructions (as defined in the *Hexagon Programmer's Reference Manual*), and thus cannot be grouped with other instructions in an instruction packet.
- They are generally implemented as Hexagon traps which switch the processor to its Monitor mode.

A.2.1 VMCACHE

Perform cache maintenance operations which are either impossible or inefficient using the native cache management instructions.

Syntax

```
vmcache
```

Registers

Register	In/Out	Description
R0	In	Cache operation to perform: #0: ICKILL - Nondestructive Invalidate of entire instruction cache #1: DCKILL - Nondestructive invalidate of entire data cache #2: L2KILL - Nondestructive invalidate of entire L2 data cache #3: DCCLEANINVA - Nondestructive invalidate of the range of virtual addresses starting at the address in R1 for the number of bytes in R2 through the full D-cache hierarchy. #4: ICINVA - Nondestructive invalidate of the range of virtual addresses starting at the address in R1 for the number of bytes in R2 through the full I-cache hierarchy. #5: IDSYNC - Ensures that Instruction cache can observe all data in the specified range; similar to DCCLEANINVA followed by ICINVA. other: Reserved (No-Op)
	Out	0: Operation successful -1: Operation unsuccessful
R1	In	Starting virtual address for range-based operations
R2	In	Byte count for range-based operations

Type: JR (slot 2)

Exceptions

- Store protection exception on range-based operations if virtual address range contains pages without write permission.
- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								**23**	**22**						**16**	**15**	**14**	**13**	**12**				**8**	**7**		**5**	**4**			**2**	**1**	**0**
VM (trap1)									-							Parse		-	VM Group 1					-			vmcache				-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	1	-	-	-	1	0	1	-	-

A.2.2 VMCLRMAP

Flush possible stale virtual mapping.

Syntax

```
vmclrmap
```

Registers

Register	In/Out	Description
R0	In	Starting virtual address. Purge MMU state for the virtual address range starting at the address in R0, and covering the number of bytes in R1. NOTE - This would typically follow a modification to an active page table.
	Out	0: Operation successful -1: Operation unsuccessful
R1	In	Byte count

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 1					-			vmclrmap			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	1	-	-	-	0	1	0	-	-

A.2.3 VMGETIE

Get virtual processor interrupt enable value.

Syntax

```
vmgetie
```

Registers

Register	In/Out	Description
R0	Out	0: Interrupts disabled on virtual processor 1: Interrupts enabled on virtual processor

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 0					-			vmgetie			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	P	P	-	0	0	0	0	0	-	-	-	1	0	0	-	-

A.2.4 VMINTOP

Operate on virtual interrupt controller.

Syntax

```
vmintop
```

Registers

Register	In/Out	Description
R0	In	Interrupt operation to perform: 0: VMINTOP_NOP: Do nothing 1: VMINTOP_GLOBEN: Global Enable 2: VMINTOP_GLOBIDIS: Global Disable 3: VMINTOP_LOCEN: Local Enable 4: VMINTOP_LOCDIS: Local Disable 5: VMINTOP_AFFINITY: Set Interrupt Affinity 6: VMINTOP_GET: Take Next Interrupt 7: VMINTOP_PEEK: Query which interrupt is next, without taking 8: VMINTOP_STATUS: For an interrupt, return pending and enablement status 9: VMINTOP_POST: Post an interrupt into interrupt controller 10: VMINTOP_CLEAR: Clear an interrupt without taking it
	Out	Non-zero value for failure for operations 1-5 and 9-10. R0=6:GET: returns 0 if interrupt taken, -1 if not taken. R0=7: PEEK: returns highest priority pending/enable interrupt, otherwise -1. R0=8: STATUS: 0 if pending, 1 if local enable. 2: global enable. -1 on failure.
R1	In	Interrupt that the operation acts upon.
R2	In	Only used if R0 = 5: the virtual processor to take the interrupt.

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								**23**	**22**						**16**	**15**	**14**	**13**	**12**				**8**	**7**		**5**	**4**			**2**	**1**	**0**
VM (trap1)									-							Parse		-	VM Group 0					-			vmintop				-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	0	-	-	-	1	0	1		-	-

A.2.5 VMGETTIME

Get 64-bit virtual processor timestamp.

Syntax

```
vmgettime
```

Registers

Register	In/Out	Description
R0	Out	Least-significant 32 bits of virtual processor timestamp
R1	Out	Most-significant 32 bits of virtual processor timestamp

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 1					-			vmgettime			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	1	-	-	-	1	1	0	-	-

A.2.6 VMNEWMAP

Establish new logical-to-virtual memory map.

The virtual machine's MMU is reprogrammed to discard the previous virtual-physical mappings in effect prior to the `vmnewmap` instruction and use the mapping described by the page tables whose segment table (L1 page table) begins at the address in R0. If the translations map logical addresses that are outside the virtual machine's legal logical address space, it is implementation specific whether the operation fails, returning a non-zero error code, or whether a machine check event will be generated when an illegal entry is encountered at run time. VMM implementations may not implement all translation table types. An unsupported translation type will result in an error and translations will not be changed.

Syntax

```
vmnewmap
```

Registers

Register	In/Out	Description
R0	In	**Logical** address of new MMU segment table.
	Out	0 if operation was successful, otherwise an implementation-specific negative error code.
R1	In	0: Linear list of translations 1: Set of page tables

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode
- Machine check event if invalid page table subsequently detected at run time by a virtual machine implementation which does not do full page table validation at `vmnewmap` time.

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.
- The virtual machine instruction following the `vmnewmap` operation will be the instruction at the address following the `vmnewmap's` address in the original map, as translated by the new map. Care must be taken in replacing the mapping of guest OS pages containing `vmnewmap` instructions.

Encoding

31								**23**	**22**						**16**	**15**	**14**	**13**	**12**				**8**	**7**		**5**	**4**			**2**	**1**	**0**
VM (trap1)									-							Parse		-	VM Group 1					-			vmnewmap				-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	1	-	-	-	0	1	1	-	-

A.2.7 VMRTE

Return from event service

- PC = G0 (GELR)
- Interrupt Enable State = G1[30] (GSR.IE)
- User Mode = G1[31] (GSR.UM)
- if (User Mode) swap(R29,G2) (GOSP)

Syntax

```
vmrte
```

Registers

Register	In/Out	Description
R29	In/Out	Stack pointer that may be swapped with GOSP

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo Instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22						16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-							Parse		-	VM Group 0					-			vmrte			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	0	-	-	-	0	0	1	-	-

A.2.8 VMSETIE

Set virtual processor interrupt enable value.

Syntax

```
vmsetie
```

Registers

Register	In/Out	Description
R0	In	1 (or any odd value): Enable interrupts to the virtual processor 0 (or any even value): Disable interrupts to the virtual processor
	Out	0: Interrupts were previously disabled on the virtual processor 1: Interrupts were previously enabled on the virtual processor

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 0					-			vmsetie			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	0	-	-	-	0	1	1	-	-

A.2.9 VMSETTIME

Set virtual processor timestamp value.

Syntax

```
vmsettime
```

Registers

Register	In/Out	Description
R0	In	Least-significant 32 bits of virtual processor timestamp
R1	In	Most-significant 32 bits of virtual processor timestamp

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo Instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 1				-			vmsettime			-		
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	1	-	-	-	1	1	1	-	-

A.2.10 VMSETVEC

Set event handler vector table for events.

Syntax

```
vmsetvec
```

Registers

Register	In/Out	Description
R0	In	Vector table address
	Out	0: Success -1: Failure

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo Instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 0					-			vmsetvec			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	0	-	-	-	0	1	0	-	-

A.2.11 VMSTART

Start a new, concurrently-executing virtual processor.

If resources are available, a new virtual processor is allocated to begin execution in guest mode with the instruction address contained in R0. The stack pointer (R29) of the new virtual processor is set to the value of R1 of the virtual processor executing the `vmstart`. The virtual processor number of the new virtual processor is returned in R0 of the virtual processor executing the `vmstart`.

If resources are unavailable, the operation fails, and a value of -1 is returned in R0.

Syntax

```
vmstart
```

Registers

Register	In/Out	Description
R0	In	Virtual address of first instruction to be executed by new virtual processor
	Out	If successful, the unique identifier number of the newly created virtual processor. If unsuccessful, a value of -1.
R1	In	Virtual address of initial top of stack for new virtual processor

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.
- If the initial instruction virtual address is not valid, an execute protection violation exception event is taken by the **new virtual processor.**

Encoding

31								**23**	**22**					**16**	**15**	**14**	**13**	**12**				**8**	**7**		**5**	**4**		**2**	**1**	**0**
VM (trap1)									-						Parse		-	VM Group 2					-			vmstart			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	P	P	-	0	0	0	1	0	-	-	-	0	1	0	-	-

A.2.12 VMSTOP

Terminate virtual processor execution.

The virtual processor executing the `vmstop` instruction ceases execution, and its resources are made available for future `vmstart` operations.

Syntax

```
vmstop
```

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 2				-			vmstop			-		
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	P	P	-	0	0	0	1	0	-	-	-	0	1	1	-	-

A.2.13 VMVERSION

Request and obtain the virtual machine version

Syntax

```
vmversion
```

Registers

Register	In/Out	Description
R0	In	Requested version for HVM to conform to
	Out	Virtual machine version that is conformed to
R1	In	Most-significant 32 bits of virtual processor timestamp

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.
- The version corresponding to this spec is `0x0000_0700`.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 0					-			vmversion			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	0	0	-	-	-	0	0	0	-	-

A.2.14 VMVPID

Get virtual processor identification number of the current processor.

Syntax

```
vmvpid
```

Registers

Register	In/Out	Description
R0	Out	Virtual processor identification number of current processor

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22						16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-							Parse		-	VM Group 2					-			vmvpid			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	1	0	-	-	-	1	0	0	-	-

A.2.15 VMWAIT

Wait on an interrupt event.

The virtual processor issuing the `vmwait` suspends execution until a configured and unmasked interrupt request is presented to the virtual processor. If interrupts are disabled, execution resumes with the instruction following the `vmwait`. If interrupts are enabled, the interrupt is serviced with the GELR value in the event record containing the address of the instruction following the `vmwait`, so that normal execution resumes after a `vmrte` is executed with the interrupt event record on the top of the stack. In either case, when execution resumes, R0 contains the interrupt number of the interrupt which terminated the wait. If more than one configured and unmasked interrupt is detected before execution resumes, it is implementation dependent which of the interrupts will be represented in R0.

Syntax

```
vmwait
```

Registers

Register	In/Out	Description
R0	Out	Interrupt number associated with the interrupt request that terminated the `vmwait`.

Type: JR (slot 2)

Type

- Privilege violation exception if executed in User mode

Notes

- This is a solo Instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22					16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-						Parse		-	VM Group 2				-			vmwait			-		
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	1	0	-	-	-	0	0	0	-	-

A.2.16 VMYIELD

Voluntarily reschedule virtual processor resources.

If other virtual processors at the same or higher priority are runnable but waiting on processor resources, suspend execution of the issuing virtual processor and schedule the virtual processor with the highest priority.

Syntax

```
vmyield
```

Type: JR (slot 2)

Exceptions

- Privilege violation exception if executed in User mode

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31								23	22						16	15	14	13	12				8	7		5	4		2	1	0
VM (trap1)									-							Parse		-	VM Group 2					-			vmyield			-	
0	1	0	1	0	1	0	0	1	-	-	-	-	-	-	-	P	P	-	0	0	0	1	0	-	-	-	0	0	1	-	-

B Determining HVM Environment

B.1 Overview

Guest-mode software can poll the VMM for the current HVM version to determine if they are version-compatible.

B.2 Accessing environment version

When Guest-mode software begins execution, it may need an efficient way to determine the HVM environment provided by the VMM. This can be done with the virtual instruction `vmversion`, which returns a value specifying the virtual machine environment version that is supported by the VMM.

This enables the guest software to request a virtual machine environment version that it has been verified to be compatible with.

This feature enables the VMM to support guests that may only support an older HVM specification. Alternatively, it enables guests to support Monitors that only support older HVM specifications.

NOTE Further communication about the system configuration will likely be beneficial for the guest, but the definition of this feature is TBD.

The version number associated with the current VM specification is `0x0000_0700`.

B.3 Virtual instructions for determining environment

Table B-1 summarizes the virtual instructions used for determining the HVM environment.

Table B-1 Virtual instructions for determining environment

Instruction	Description
`vmversion`	Return the virtual machine version

For details on these instructions see Appendix A.