NVIDIA Compute

PTX: Parallel Thread Execution
ISA Version 1.0

Release 1.0

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, GeForce, and NVIDIA Quadro are registered trademarks of NVIDIA Corporation. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

HISTORY OF MAJOR REVISIONS

Version	Date	Changes
1.0	6/12/2007	PTX ISA Version 1.0, release 1.0.

Table of Contents

Section 1. Introduction

1.1. Data-Parallel Computing using GPUs

This document describes PTX, a low-level *parallel thread execution* virtual machine (VM) and virtual instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing *device*.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets such as arrays can use a data-parallel programming model to speed up the computations. Data-parallel mapping is efficient for SIMD, vector, and highly multi-threaded parallel architectures. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. Many compute-intensive applications map well to data-parallel processing. In general, all algorithms that can be formulated as parallel computations operating over datasets are good candidates for acceleration by data-parallel processing.

PTX defines a virtual machine and virtual ISA for general purpose parallel thread execution. PTX programs are translated at install time to the target hardware instruction set. The PTX to GPU translator and driver enables NVIDIA GPUs to be used as programmable parallel computers.

1.2. Goals of PTX

PTX provides a stable programming model and instruction set for general purpose parallel programming. It is designed to be efficient on NVIDIA GPUs supporting the computation features defined for G80 and subsequent GPUs. High level language compilers for languages such as C and C++ generate PTX instructions, which are optimized for and translated to native target-architecture instructions.

The goals for PTX include the following:

- Provide a stable virtual ISA and VM that spans multiple GPU generations.
- Achieve performance in compiled applications comparable to native GPU performance.
- Provide a machine-independent ISA for C/C++ and other compilers to target.
- Provide a code distribution ISA for application and middleware developers.
- Provide a common source-level ISA for optimizing code generators and translators, which map PTX to specific target machines.

- Programmability – facilitate hand-coding of libraries, performance kernels, and architecture tests.
- Scalability – VM programming model spans GPU sizes from single unit to many parallel units.
- Provide a relatively low-level ISA and machine model that can be usefully thought of as representing the target GPU architecture.
- VM and virtual ISA will become publicly visible.
- Component of the NV Compute product.
- Compatibility – version 1 programs execute on later translators.
- Sufficient quality and usability to evolve into an industry standard.

1.3. The Document's Structure

This document is organized in the following way:

Section 2 outlines the programming model.

Section 3 gives an overview of the PTX virtual machine model.

Section 4 describes the basic syntax of the PTX language.

Section 5 describes state spaces, types, and variable declarations.

Section 6 describes instruction operands.

Section 7 describes the instruction set.

Section 8 lists special registers.

Section 9 lists the assembly directives supported in PTX.

Section 10 provides notes for Release 1.0 of PTX Version 1.0.

Section 2. Programming Model

2.1. A Highly Multithreaded Coprocessor

The GPU is a compute device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU, or host: In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device.

More precisely, a portion of an application that is executed many times, but independently on different data, can be isolated into a function that is executed on the GPU as many different threads. To that effect, such a function is compiled to the PTX instruction set and the resulting kernel is translated at install time to the target GPU instruction set.

2.2. Thread Batching

The batch of threads that executes a kernel is organized as a *grid* of *cooperative thread arrays* as described in this section and illustrated in Figure 1.

2.2.1. Cooperative Thread Arrays

The Parallel Thread Execution (PTX) programming model is explicitly parallel – a PTX program specifies the execution of a given thread of a parallel thread array. A *Cooperative Thread Array*, or *CTA*, is an array of threads that execute a kernel concurrently or in parallel.

Threads within a CTA can communicate with each other. To coordinate the communication of the threads within the CTA, one can specify synchronization points, where threads are suspended until they all reach the synchronization point.

Each thread has a unique *thread id* (*tid*) within the CTA. Programs use a data parallel decomposition to partition inputs, work, and results across the threads of the CTA. Each CTA thread uses its tid to determine its assigned role, assign specific input and output position, compute addresses, and select work to perform. The tid is a 3-component vector, *tid.0, tid.1,* and *tid.2*, that specifies the thread's position within a 1D, 2D, or 3D CTA. Alternate component names are *tid.x*, *tid.y*, and *tid.z*. Each tid component ranges from 0 up to the number of thread id's in that CTA dimension.

Each CTA has a a 1D, 2D, or 3D shape, specified by a 3-component vector, *ntid*, which specifies the number of threads in each CTA dimension. The ntid components are accessible as *ntid.0*, *ntid.1*, and *ntid.2*.

Threads within a CTA execute in SIMD fashion in groups called *warps*. A warp is a maximal subset of threads from a single CTA, such that the threads execute the same

instructions at the same time. Threads within a warp are sequentially numbered. The warp size is a machine-dependent constant. Typically, a warp has 16 or 32 threads. Some applications may be able to maximize performance with knowledge of the warp size, so PTX includes a run-time immediate constant, WARP_SZ, which may be used in any instruction where an immediate operand is allowed.

2.2.2. Grid of Cooperative Thread Arrays

There is a maximum number of threads that a CTA can contain. However, CTAs that execute the same kernel can be batched together into a *grid* of CTAs, so that the total number of threads that can be launched in a single kernel invocation is very large. This comes at the expense of reduced thread communication and synchronization, because threads in different CTAs cannot communicate and synchronize with each other.

Multiple CTAs may execute concurrently and in parallel, or sequentially, depending on the platform. Each CTA has a unique *CTA id* (*ctaid*) within a grid of CTAs. Each grid of CTAs has a 1D, 2D , or 3D shape specified by the parameter *nctaid***.** Each grid also has a unique temporal *grid id* (*gridid***)**. Threads may read and use these values through predefined, read-only special registers *%tid*, *%ntid*, *%ctaid*, *%nctaid*, and *%gridid*.

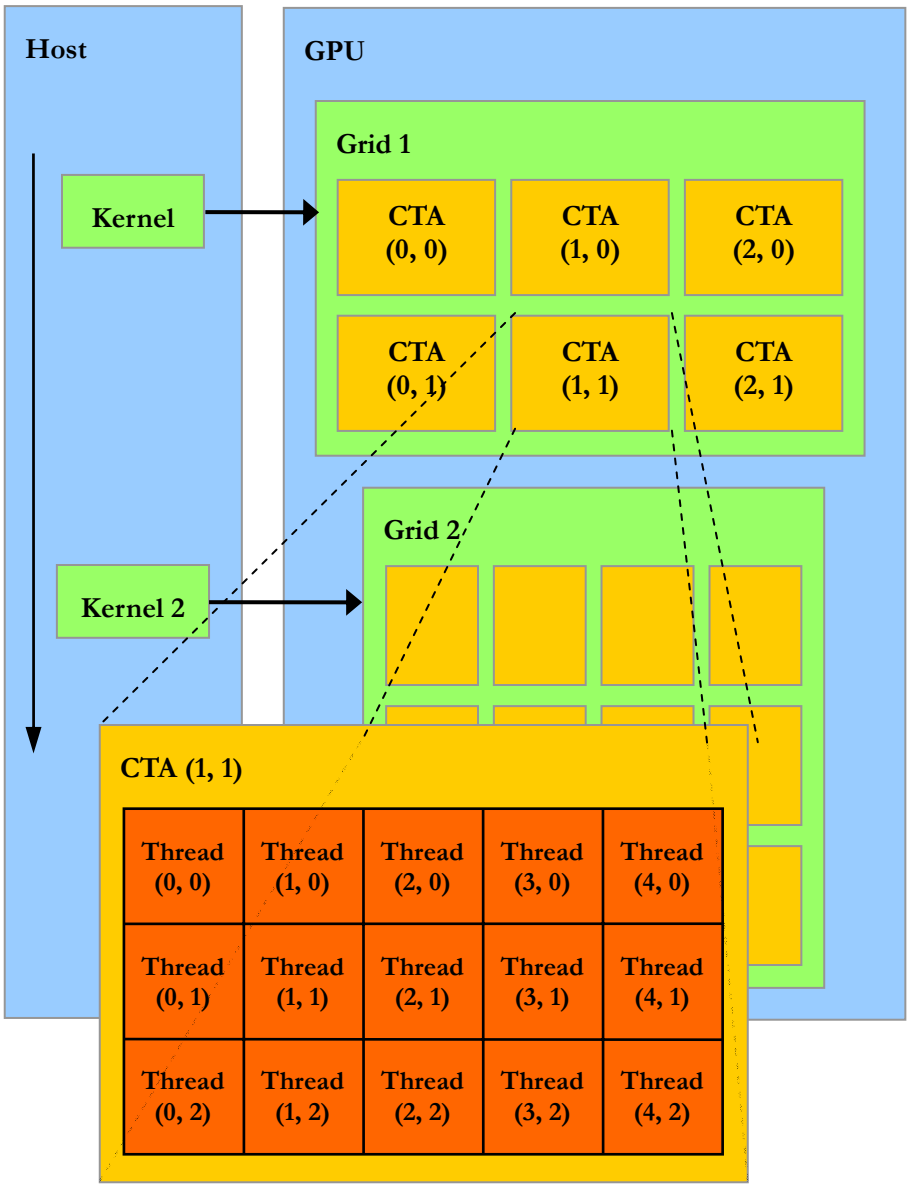

Figure 1: Thread Batching

The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of CTAs.

Section 3. The Parallel Thread Execution Machine Model

3.1. A Set of SIMD Multiprocessors with On-Chip Shared Memory

The PTX machine model is implemented as a set of multiprocessors as illustrated in Figure 2. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data.

Both the host and the device maintain their own local memory, referred to as *host memory* and *device memory*, respectively. The device memory may be mapped and read or written by the host, or, for more efficient transfer, copied from the host memory through optimized API calls that utilize the device's high-performance Direct Memory Access (DMA) engine.

Each multiprocessor has *on-chip memory* of the four following types:

- One set of local 32-bit *registers* per processor,
- *Shared memory* that is shared by all the processors,
- A read-only *constant cache* that is shared by all the processors and speeds up reads from the constant memory, which is a read-only region of the device memory,
- A read-only *texture cache* that is shared by all the processors and dedicated to texture sampling.

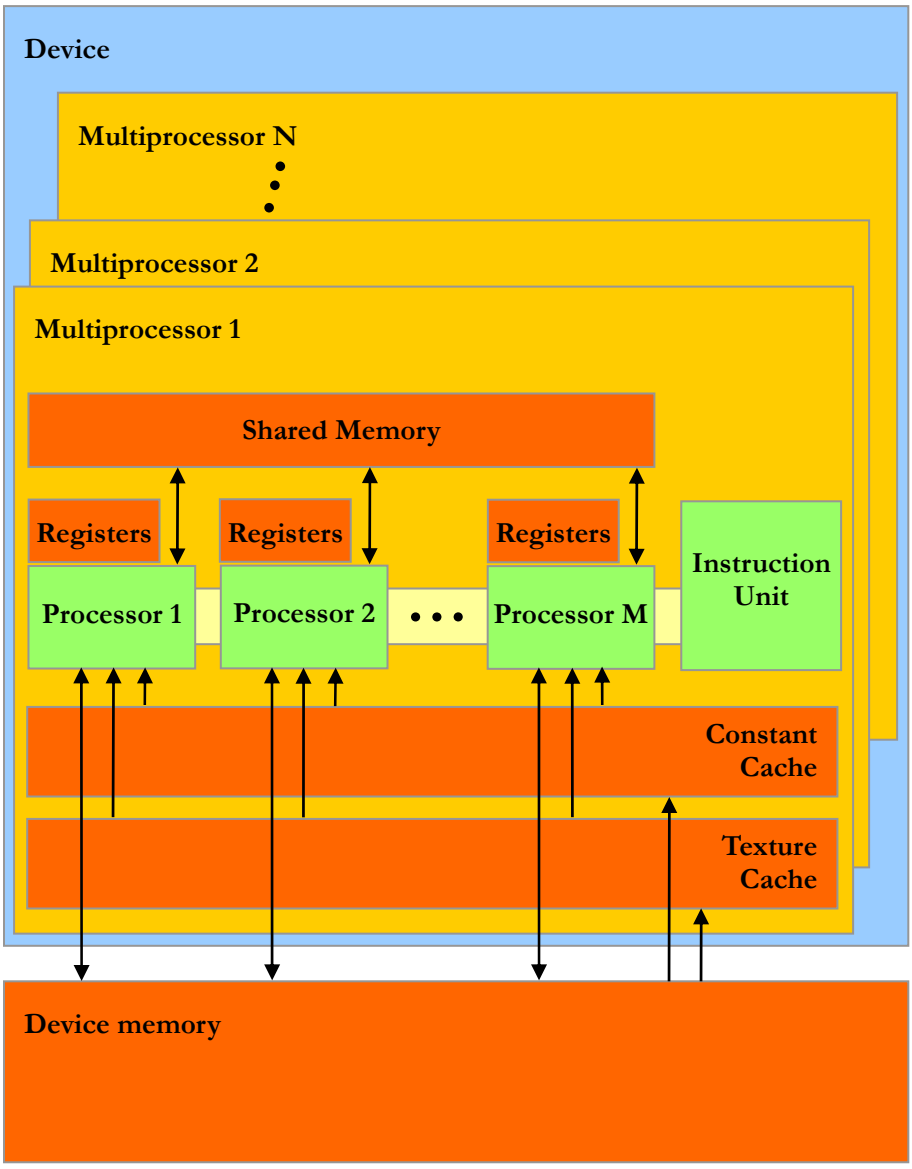

Figure 2: Machine Model

A set of SIMD multiprocessors with on-chip shared memory.

3.2. Execution Model

A grid of CTAs is executed on the device by executing one or more CTAs on each multiprocessor using time slicing: Each CTA is split into SIMD groups of threads called *warps*; each of these warps contains the same number of threads, called the *warp size*, and is executed by the multiprocessor in a SIMD fashion; a thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor's computational resources.

The way a CTA is split into warps is always the same; each warp contains threads of consecutive, increasing thread indices with the first warp containing thread 0.

A CTA is processed by only one multiprocessor, so that threads within a CTA can use the on-chip shared memory to efficiently share data among them. More precisely, threads can perform general reads from and writes to the on-chip shared memory through a per-CTA shared memory partition and coordinate these memory accesses through synchronization mechanisms.

A multiprocessor can process several CTAs concurrently by partitioning its resources (e.g. registers and shared memory) among them.

Threads can access several other memory partitions:

- Threads can perform general cached reads from the constant memory through a per-grid constant memory partition.
- Threads can perform general non-cached reads from and writes to the device memory through two device memory partitions: a per-thread *local memory partition* and a per-grid *global memory partition*.
- At last, another way to perform general cached reads from the device memory is through texture sampling.

This memory model is illustrated in Figure 3.

The issue order of the CTAs within a grid is not defined and there is no synchronization mechanism between CTAs, so threads from two different CTAs of the same grid cannot safely communicate with each other through the device memory.

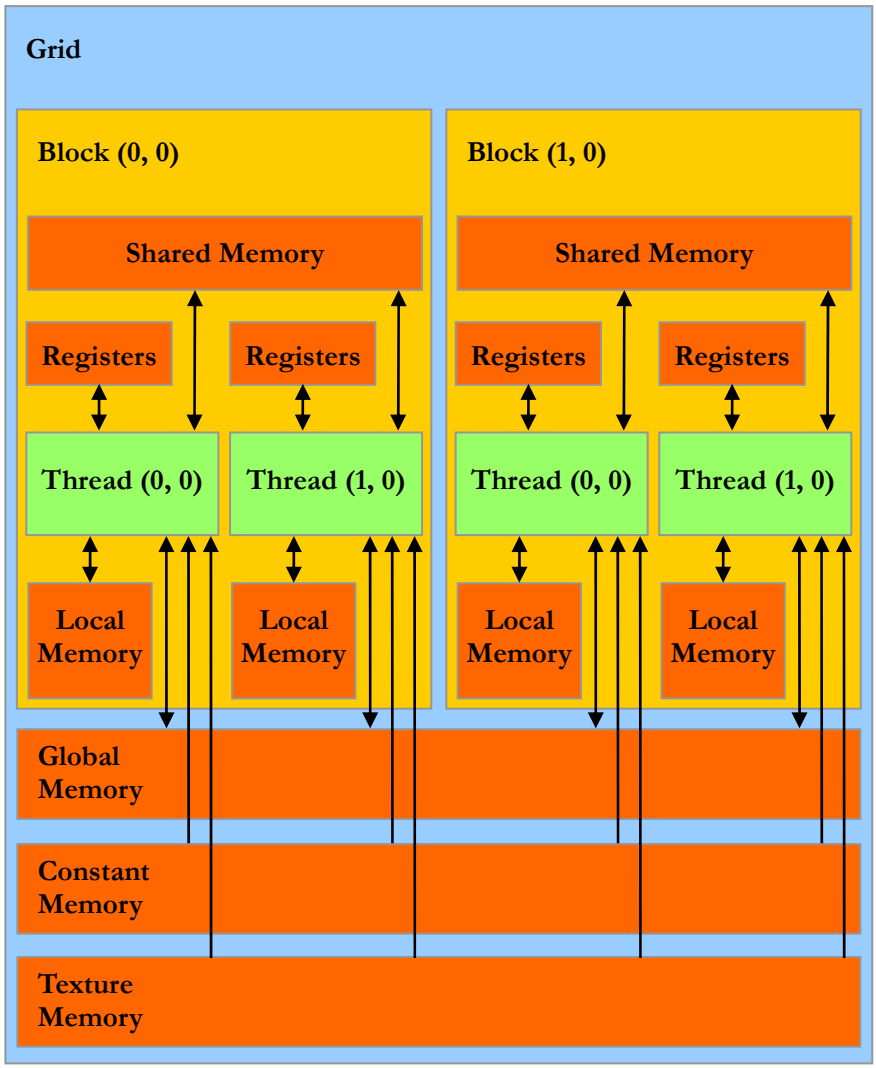

Figure 3: Memory Model

Shared memory and registers are on chip. Texture memory, constant memory, local memory, and global memory are in device memory. Reads from texture memory and constant memory are cached. Reads from and writes to local and global memory are not cached.

Section 4. Syntax

PTX programs are a collection of text source files. PTX source files have an assembly-language style syntax with instruction operation codes and operands. Pseudo-operations specify symbol and addressing management. The **ptxas** program assembles PTX source files to produce corresponding binary object files.

4.1. Source Format

Source files are ASCII text. Lines are separated by the newline character ('\n').

All whitespace characters are equivalent; whitespace is ignored except for its use in separating tokens in the language.

The C preprocessor **cpp** may be used to process PTX source files. Lines beginning with # are preprocessor directives. The following are common preprocessor directives:

```
#include, #define, #if, #ifdef, #else, #endif, #line, #file
```

"C: A Reference Manual" by Harbison and Steele provides a good description of the C preprocessor.

PTX is case sensitive and uses lowercase for keywords.

Each PTX file must begin with a .version directive specifying the PTX language version, followed by a .target directive specifying the target architecture assumed. See Section 9 for a more information on these directives.

4.2. Comments

Comments in PTX follow C/C++ syntax, using non-nested **/*** and ***/** for comments that may span multiple lines, and using **//** to begin a comment that extends to the end of the current line.

Comments in PTX are treated as whitespace.

4.3. Statements

A PTX statement is either a directive or an instruction. Statements begin with an optional label and end with a semicolon.

Examples:

```
        .reg     .b32 r1, r2;
        .global  .f32  array[N];

start:  mov.b32   r1, %tid.0;
        shl.b32   r1, r1, 2;        // shift thread id by 2 bits
        ld.b32    r2, array[r1];    // thread[tid] gets array[tid]
        add.f32   r2, r2, 0.5;      // add 1/2
```

4.3.1. Directive Statements

Directive keywords begin with a dot, so no conflict is possible with user-defined identifiers. The directives in PTX are listed in Table 1 and described in Section 9.

.byte	.func	.reg	.target
.const	.global	.section	.tex
.entry	.local	.shared	.version
.extern	.loc	.sreg	.visible
.file	.param	.surf	

Table 1: Directives

4.3.2. Instruction Statements

Instructions are formed from an instruction opcode followed by a comma-separated list of zero or more operands, and terminated with a semicolon. Operands may be register variables, constant expressions, address expressions, or label names. Instructions have an optional guard predicate which controls conditional execution. The guard predicate follows the optional label and precedes the opcode, and is written as **@p**, where p is a predicate register. The guard predicate may be optionally negated, written as **@!p**.

The destination operand is first, followed by source operands.

Instruction keywords are listed in Table 2. All instruction keywords are reserved tokens in PTX.

abs	div	max	ret	sub
add	dot	membar	rsqrt	tex
and	ex2	min	sad	trap
atom	exit	mov	selp	vote
bar	extract	mul	set	vred
bra	frc	mul24	setp	xor
brkpt	insert	neg	shl	
call	ld	nop	shr	
cnot	lg2	not	sin	
cos	mad	or	slct	
cross	mad24	rcp	sqrt	
cvt	mag	rem	st	

Table 2: Reserved Instruction Keywords

4.4. Identifiers

User-defined identifiers follow extended C++ rules: they start with an alphabetic character, underscore, dollar sign, or percentage sign ([**A-Za-z_$%**]) and are followed by zero or more alphanumeric, underscore, or dollar sign characters ([**A-Za-z_$**]).

Many high-level languages such as C and C++ follow similar rules for identifier names, except that the percentage sign is not allowed. PTX allows the percentage sign as the first character of an identifier. The percentage sign can be used to avoid name conflicts, e.g. between user-defined variable names and compiler-generated names.

PTX predefines a small number of special registers that begin with the percentage sign, listed in Table 3.

%ctaid	%clock	%pm2
%gridid	%physid	%pm3
%nctaid	%pm0	%tid
%ntid	%pm1	

Table 3: Predefined identifiers

4.5. Immediate Constants

Immediate constants in PTX are restricted to integer and floating-point types.

4.5.1. Integer Immediate Constants

Integer immediate constants may be written in decimal, hexadecimal, octal, or binary notation.

Decimal constants begin with a nonzero digit followed by zero or more digits (**0-9**).

Hexadecimal constants begin with **0x** or **0X** followed by one or more hex digits (from the set [**0-9a-fA-F**]).

Octal constants begin with zero **0** followed by zero or more octal digits (**0-7**).

Binary constants begin with **0b** or **0B** followed by one or more binary digits (**01**).

4.5.2. Floating-point Immediate Constants

Floating-point immediate constants may be written with an optional decimal point and an optional signed exponent. Unlike C and C++, there is no suffix letter to specify size (e.g. float or double).

PTX includes a second representation of floating-point constants, where the exact machine representation is given as a hexadecimal constant. For 64-bit floating point values, the constant begins with **0d** or **0D** followed by 16 hex digits. For 32-bit floating point values, the constant begins with **0f** or **0F** followed by 8 hex digits.

4.5.3. Predicate Immediate Constants

Predicate immediate constants for the Boolean values **TRUE** and **FALSE** are written as binary digits **1** and **0**, respectively.

4.5.4. Constant Expressions

Constant expressions are evaluated at compile time to form simple values for use in immediate operands and addressing expressions. Both integer and floating-point constant expressions are supported, however, note that floating-point constant expressions may evaluate to a different value than would be computed on the target architecture, since the compiler may evaluate the expression using greater precision than the target architecture.

Constant expressions are formed from lexical constants, basic arithmetic operators (addition, subtraction, multiplication, division), and parentheses. Integer constant expressions may include remainder (**%**), shift operators (**<<** and **>>**), and logical operators (**&**, **|**, and **^**).

The meaning of operators in PTX is the same as in C or C++.

Section 5. State Spaces, Types, and Variables

While the specific resources available in a given target GPU will vary, the kinds of resources will be common across platforms, and these resources are abstracted in PTX through state spaces and data types.

5.1. State Spaces

A *state space* is a storage area with particular characteristics. All variables reside in some state space. The characteristics of a state space include its size, addressability, access speed, access rights, and level of sharing between threads.

The state spaces defined in PTX are a byproduct of parallel programming and graphics programming. The list of state spaces is shown in Table 4, and properties of state spaces are shown in Table 5.

Name	Description
.reg	Registers, fast.
.sreg	Special registers. Read-only; pre-defined; platform-specific.
.const	Per-CTA, shared, read-only memory.
.global	Global memory, shared by all threads.
.local	Local memory, private to each thread.
.param	User parameters for a program, available at CTA entry.
.shared	Addressable memory shared between threads in 1 CTA.
.surf	Global surface memory.
.tex	Global texture memory.

Table 4: State spaces

Name	Addressible	Initializable	Access	Sharing
.reg	No	No	R/W	per-thread
.sreg	No	No	RO	per-CTA
.const	Yes	Yes	RO	per-grid
.global	Yes	Yes	R/W	Context
.local	Yes	No	R/W	per-thread
.param	Yes	No	RO	per-grid
.shared	Yes	No	R/W	per-CTA
.surf	via LD/ST, SURF instructions	Yes	R/W	Context
.tex	via TEX instruction	Yes	RO	Context

Table 5: Properties of state spaces

5.1.1. Register State Space

Registers (.reg state space) are fast storage locations. The number of registers is limited, and will vary from platform to platform. When the limit is exceeded, register variables will be spilled to memory, causing changes in performance. For each architecture, there is a recommended maximum number of registers to use.

Registers may be typed (signed integer, unsigned integer, floating point, predicate) or untyped. Register size is restricted; aside from predicate registers which are 1-bit, registers have a width of 16-, 32-, or 64-bits.

Registers differ from the other state spaces in that they are not fully addressable, i.e., it is not possible to refer to the address of a register.

Registers may have alignment boundaries required by multi-word loads and stores.

5.1.2. Special Register Space

The special register (.sreg) state space holds predefined, platform-specific registers, such as grid, CTA, and thread parameters, clock counters, and performance monitoring registers. All special registers are predefined.

5.1.3. Constant State Space

The constant (.const) state space is a read-only memory, initialized by the host. The size may be limited, and there are typically many *banks* of constant memory, denoted by an integer index. The size and number of banks are listed in the appendix for different hardware.

5.1.4. Global State Space

The global (.global) state space is memory that is accessible by all threads in a context. It is the mechanism by which different CTAs and different grids can communicate. Use ld.global, st.global, atom.global, and red.global to access global variables.

For any thread in a context, all addresses are in global memory are shared.

Global memory is not sequentially consistent. Consider the case where one thread executes the following two assignments:

```
a = a + 1;
b = b – 1;
```

If another thread sees the variable *b* change, the store operation updating *a* may still be in flight. This reiterates the kind of parallelism available in machines that run PTX. Threads must be able to do their work without waiting for other threads to do theirs, as in lock-free and wait-free style programming.

5.1.5. Local State Space

The local state space (.local) is private memory for each thread to keep its own data. It is typically standard memory with cache. The size is limited, as it must be allocated on a per-thread basis. Use ld.local and st.local to access local variables.

5.1.6. Parameter State Space

The parameter (.param) state space provides addressable user parameters to CTAs. User parameters begin at address zero, and the address space is shared across CTAs within a grid.

The location of parameter space is implementation specific. For example, in some implementations, parameter space resides in global memory. No access protection is provided between parameter and global space in this case.

5.1.7. Shared State Space

The shared (.shared) state space is a per-CTA region of memory for threads in a CTA to share data. An address in shared memory can be read and written by any thread in a CTA. Use ld.shared and st.shared to access shared variables.

Shared memory typically has some optimizations to support the sharing. One example is broadcast; where all threads read from the same address. Another is sequential access from sequential threads.

5.1.8. Texture State Space

The texture (.tex) state space is global memory for the texture instructions. It is shared by all threads in a context.

The GPU hardware has a fixed number of texture bindings that can be accessed within a single program (typically 128). The .tex[i] directive will bind the named texture memory variable to the hardware texture id 'i'. If no id number is given, PTX will assign

texture id's sequentially, beginning with zero. Multiple names may be bound to the same physical texture id. An error is generated only if the texture id assigned is out of the physical texture id range (e.g., 0..127).

Texture memory is read-only.

Example:

```
.tex     tex_a;           // bound to physid 0
.tex[2]  tex_b;           // bound to physid 2
.tex     tex_c;           // bound to physid 1
.tex     tex_d;           // bound to physid 2
.tex[42] tex_e;           // bound to physid 42
.tex     tex_f;           // bound to physid 3
```

5.1.9. Surface State Space

The surface (.surf) state space is similar to global memory, but is 2D in nature. It takes a 2D address (*i* and *j* components), and with respect to cache, spatial locality generally works well in a 2D neighborhood. This allows tiled decompositions to perform quite well.

5.2. Types

5.2.1. Fundamental Types

In PTX, the fundamental types reflect the native data types supported by the target architectures. A fundamental type specifies both a basic type and a size. Register variables are always of a fundamental type, and instructions operate on these types. The same type-size specifiers are used for both variable definitions and for typing instructions, so their names are intentionally short.

The following table lists the fundamental type specifiers for each basic type:

Basic Type	Fundamental Type Specifiers
Signed integer	.s8, .s16, .s32, .s64
Unsigned integer	.u8, .u16, .u32, .u64
Floating-point	.f16, .f32, .f64
Bits (untyped)	.b8, .b16, .b32, .b64
Predicate	.pred

Most instructions have one or more type specifiers, needed to fully specify instruction behavior. Operand types and sizes are checked against instruction types for compatibility.

Two fundamental types are compatible if they have the same basic type and are the same size. Signed and unsigned integer types are compatible if they have the same size. The bit-size type is compatible with any fundamental type having the same size.

In principle, all variables could be declared using only bit-size types, but typed variables enhance program readability and allow for better operand type checking.

5.2.2. Restricted Use of Sub-word Sizes

The .u8 and .s8 types are restricted to **ld**, **st**, and **cvt** instructions. The **ld** and **st** instructions also accept .b8 type. Byte-size integer load instructions zero- or sign-extended the value to the size of the destination register.

The .f16 floating-point type is allowed only in conversions to and from .f32 and .f64 types. All floating-point instructions operate only on .f32 and .f64 types.

5.3. Variables

In PTX, a variable declaration describes both the variable's type and its state space. In addition to fundamental types, PTX supports types for aggregate objects such as vectors, arrays, structures and unions.

5.3.1. Variable Declarations

All storage for data is specified with variable declarations. Every variable must reside in one of the state spaces enumerated in the previous section.

A variable declaration names the space in which the variable resides, its type and size, its name, an optional array size, an optional initializer, and an optional fixed address for the variable.

Examples:

```
.global .u32 loc;
.reg .s32 i = 0;
.shared .f32 bias[] = {-1.0, 1.0};
.local .u8 bg[4] = {0, 0, 0, 0};
.reg .v3 .f32 accel;
.struct float4 { .f32[4] v };
.global float4 coord;
```

Note that texture and surface variables do not have an associated type and size.

5.3.2. Vectors

Limited-length vector types are supported. Vectors of length 2, 3, and 4 of any fundamental type can be declared by prefixing the type with .v2, .v3, or .v4. Vectors must be based on a fundamental type, and they may reside in the register space.

Examples:

```
.global .v4 .f32 V;     // a length-4 vector of floats
```

```
.shared .v2 .u16 uv;    // a length-2 vector of unsigned ints
.reg .v4 .pred vpred;   // a vector of predicates registers
```

5.3.3. Array Declarations

Array declarations are provided to allow the programmer to reserve space. To declare an array, the variable name is followed with dimensional declarations similar to fixed-size array declarations in C. The size of the dimension is either a constant expression, or is left empty, being determined by an array initializer. Here are some examples:

```
.local .u16 kernel[19][19];
.shared .u8 mailbox[128];
.shared .s32 offset[][] = { {-1, 0}, {0, -1}, {1, 0}, {0, 1} };
```

The size of the array specifies how many elements should be reserved. For the *kernel* declaration above, 19*19 (361) halfwords are reserved (722 bytes).

5.3.4. Structures and Unions

A structure definition specifies a sequence of fields (consisting of a type/size and a name) as a block of memory. This is analogous to the structures in C. Once defined, the structure can be used as a type designator in subsequent variable declarations.

Example:

```
.struct somestruct { .s32 i; .s32 j; .f32 x; .f32 y; };
.global somestruct p;
.reg .b32 ptr;
…
ld.s32  r0, [p.x];
mov.b32 ptr, p;    // get address of structure p
```

Unions definitions use the same syntax as struct definitions, with the keyword **.struct** replaced by **.union**. The difference between a struct and a union is that in a struct, the fields are laid out sequentially in memory, while in a union, the fields all use the same memory. Unions provide a way to reuse memory in a relatively type-safe manner. Here is an example that provides storage for a float or an integer:

```
.union intOrFloat { .s32 i; .f32 f; };
```

Structure and union declarations may be nested. The shortcut syntax of C++ with anonymous unions is also supported.

5.3.5. Initializers

All declarations may specify an initial value for the variable being declared (including predicates). The initializers follow the conventions of C/C++, where the variable name is followed by an equals sign and the value or values for the initial values of the variable. A scalar takes a single value; while vectors and arrays take nested lists of values inside of curly braces (the nesting matches the dimensionality of the declaration). Structures take a list of values that matches the fields in a structure.

Examples:

```
      .global .s32 n = 10;
      .shared .f16 blur_kernel[][]
                        = {{.05,.1,.05},{.1,.4,.1},{.05,.1,.05}};
      .global .v3 .u8 rgb[3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
```

Initializers for thread-private memory all initialize their variables to the same value. There is no syntax for per-thread initializers.

5.3.6. Alignment

Byte alignment of storage for all addressable variables can be specified in the variable declaration. Alignment is specified using an optional .align *byte_count* specifier immediately following the space-state specifier. The variable will be aligned to an address which is an integer multiple of *byte_count*. For arrays, structures, and unions, alignment specifies the address alignment for the starting address of the entire structure, not for individual elements.

Examples:

```
// allocate array at 4-byte aligned address.  Elements are bytes.
      .const .align 4 .b8 bar[8] = {0,0,0,0,2,0,0,0};
```

Section 6. Instruction Operands

6.1. Operand Type Information

All operands in instructions have a known type from their declarations. Each operand type must be compatible with the type determined by the instruction template and instruction type. There is no automatic conversion between types.

The bit-size type is compatible with every type having the same size. Integer types of a common size are compatible with each other. Operands having type different from but compatible with the instruction type are silently cast to the instruction type.

6.2. Source Operands

The source operands are denoted in the instruction descriptions by the names **a**, **b**, and **c**. PTX describes a load-store machine, so operands for ALU instructions must all be in variables declared in the **`.reg`** register state space. For most operations, the sizes of the operands must be consistent.

The **cvt** (convert) instruction takes a variety of operand types and sizes, as its job is to convert from nearly any data type to any other data type (and size).

The **ld**, **st**, **mov**, and **cvt** instructions copy data from one location to another. Instructions **ld** and **st** move data from/to addressable state spaces to/from registers. The **mov** instruction copies data between registers.

Most instructions have an optional predicate guard that controls conditional execution, and a few instructions have additional predicate source operands. Predicate operands are denoted by the names **p**, **q**, **r**, **s**.

6.3. Destination Operands

PTX instructions that produce a single result store the result in the field denoted by **d** (for destination) in the instruction descriptions. The result operand can be any declared variable, array element, structure/union member, vector or vector element.

6.4. Using Addresses, Arrays, Vectors, Structures and Unions

Using scalar variables as operands is straightforward. The interesting capabilities begin with pointers, composite structures, and arrays.

6.4.1. Addresses as Operands

Address arithmetic is performed using integer arithmetic and logical instructions. Examples include pointer arithmetic and pointer comparisons. All addresses and address computations are byte-based; there is no support for C-style pointer arithmetic.

The `mov` instruction can be used to move the address of a variable into a pointer. Load and store operations move data between registers and locations in addressable state spaces. The syntax is similar to that used in many assembly languages, where scalar variables are simply named and addresses are de-referenced by enclosing the address expression in square brackets. Address expressions include variable names, address registers, address register plus byte offset, and immediate address expressions which evaluate at compile-time to a constant address.

Here are a few examples:

```
.shared .u16 x;
.reg .u16 r0;
.global .v4 .f16 V;
.reg .v4 .f16 W;
.const .s32 tbl[256];
.reg .b32 p;
.reg .s32 q;

ld.u16    r0,[x];
ld.v4.f16 W, [V];
ld.s32    q, [tbl+12];
mov.b32   p, tbl;
```

6.4.2. Arrays as Operands

Arrays of all types can be declared, and the identifier becomes an address constant in the space where the array is declared. The size of the array is a constant in the program.

Array elements can be accessed using an explicitly calculated byte address, or by indexing into the array using square-bracket notation. The expression within square brackets is either a constant integer, a register variable, or a simple "register with constant offset" expression, where the offset is a constant expression that is either added or subtracted from a register variable. If more complicated indexing is desired, it must be written as an address calculation prior to use. Examples are

```
ld.u32  s, a[0];
ld.u32  s, a[N-1];
mov.u32 s, a[1];        // move address of a[1] into s
```

6.4.3. Vectors as Operands

Vectors can be treated as a collection of elements simply by naming them. Vector variables can typically replace scalar variables in most PTX instructions, and the meaning is to perform the operation on an element-by-element basis.

```
.reg .v4 .f16 v1, v2, v3;
add.v4.s32 v3, v2, v1;
```

Vector elements can be extracted from the vector with the suffixes .0, .1, .2, and .3 or .x, .y, .z and .w suffixes, as well as the typical color fields .r, .g, .b and .a.

Vectors can be swizzled or reordered with swizzling suffixes, which are a combination of the digits or characters that represent the elements of a vector (0123, xyzw, rgba). The swizzling suffixes allow arbitrary duplication and reordering of vector elements. Swizzling is allowed only in **mov** instructions, and the source and destination must be distinct.

A brace-enclosed list is used for pattern matching to pull apart vectors. Wide loads and stores can be specified to multiple targets using vector loads, specifying multiple scalars within the brace-enclosed list. Here are some examples:

```
.reg .v3 .f32 V;
.reg .f32 a, b, c;
mov.v3.f32 {a,b,c}, V;
```

Vector loads and stores can be used to implement wide loads and stores, which may improve memory performance. The registers in the load/store operations can be a vector, or a brace-enclosed list of similarly typed scalars. Here is an example:

```
ld.v4.f32 {a,b,c,d}, [Vmem];
```

Elements in a brace-enclosed vector, say {Ra, Rb, Rc, Rd}, correspond to extracted elements as follows:

```
Ra = V.0 = V.x = V.r
Rb = V.1 = V.y = V.g
Rc = V.2 = V.z = V.b
Rd = V.3 = V.w = V.a
```

6.4.4. Structures and Unions as Operands

Structures and unions can only access their members; there are no instructions that take entire structures as operands.

6.4.5. Immediate Values as Operands

Immediate values (or constants) can be used in most instructions. Only one immediate operand is permitted in an instruction. In ALU instructions, it is typically the **b** or **c** operand. In load and store instructions, an immediate offset to a register or an immediate absolute address is permitted. In instruction with only one source operand, the source operand may be an immediate. The size of the immediate value may be specified with a type suffix like .u16, and defaults to the size of the instruction source operand.

For directly specifying IEEE-752 single and double precision floating point numbers, a hexadecimal value may be used as an immediate operand in floating point operations. The immediate value syntax is as follows:

```
0[fF]{hexdigit}{8}  // single-precision floating point
0[dD]{hexdigit}{16} // double-precision floating point
```

Example:

```
mov.f32  $f3, 0F3f800000;        //  1.0
```

This format may also be used when initializing variables.

6.5. Type Conversion

All operands to all arithmetic, logic, and data movement instruction must be of the same type and size, except for operations where changing the size and/or type is part of the definition of the instruction. Operands of different sizes or types must be converted prior to the operation.

6.5.1. Scalar Conversions

Table 6 below shows what precision and format the **cvt** instruction uses given operands of differing types. For example, if a **cvt.s32.u16** instruction is given a **u16** source operand and **s32** as a destination operand, the **u16** is zero-extended to **s32**.

Some of the above conversions are available at no cost (sign-extension, zero-extension, chop), while others are not (*x*2*y* conversions). The general rule is that that for integers, there is no cost converting between different sizes. All other conversions may require computation.

Conversions to floating-point that are beyond the range of floating-point numbers are represented with the maximum floating-point value (IEEE Inf for **f32** and **f64**, and ~131,000 for **f16**).

		Destination Format										
		s8	**s16**	**s32**	**s64**	**u8**	**u16**	**u32**	**u64**	**f16**	**f32**	**f64**
Source Format	**s8**	-	sext	sext	sext	-	sext	sext	sext	s2f	s2f	s2f
	s16	Chop[1]	-	sext	sext	chop[1]	-	sext	sext	s2f	s2f	s2f
	s32	Chop[1]	chop[1]	-	sext	chop[1]	chop[1]	-	sext	s2f	s2f	s2f
	s64	Chop[1]	chop[1]	chop	-	chop[1]	chop[1]	chop	-	s2f	s2f	s2f
	u8	-	zext	zext	zext	-	zext	zext	zext	u2f	u2f	u2f
	u16	Chop[1]	-	zext	zext	chop[1]	-	zext	zext	u2f	u2f	u2f
	u32	Chop[1]	chop[1]	-	zext	chop[1]	chop[1]	-	zext	u2f	u2f	u2f
	u64	Chop[1]	chop[1]	chop	-	chop[1]	chop[1]	chop	-	u2f	u2f	u2f
	f16	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	-	f2f	f2f
	f32	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	f2f	-	f2f
	f64	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	f2f	f2f	-
Notes		sext = sign extend; zext = zero-extend; chop = keep only low bits that fit; s2f = signed-to-float; f2s = float-to-signed; u2f = unsigned-to-float; f2u = float-to-unsigned; f2f = float-to-float; [1] If the destination register is wider than the destination format, the result is extended to the destination register width after chopping. The type of extension (sign or zero) is based on the destination format. For example, cvt.s16.u32 targeting a 32-bit register will first chop to 16-bits, then sign-extend to 32-bits.										

Table 6: Conversions

6.5.2. Rounding modes

Conversion instructions may specify a rounding mode. In PTX, there are four integer rounding modes and four floating-point rounding modes. The following tables summarize the rounding modes.

Mode	Description
.rn	mantissa LSB rounds to nearest even
.rz	mantissa LSB rounds towards zero
.rm	mantissa LSB rounds towards negative infinity
.rp	mantissa LSB rounds towards positive infinity

Table 7: Floating-point Rounding Modes

Mode	Description
.rni	round to nearest integer, choosing even integer if source is equidistance between two integers.
.rzi	round to nearest integer in the direction of zero
.rmi	round to nearest integer in direction of negative infinity
.rpi	round to nearest integer in direction of positive infinity

Table 8: Integer Rounding Modes

6.5.3. Vector Conversions

Conversions between scalar values and vector values are supported, allowing operations like adding the scalar value 1 to a vector. Scalar values are spread out to match the size of the vector. Short vectors are zero-extended to longer vectors, and long vectors are truncated when assigned to shorter vectors. The table below describes the conversions, where *s* is a scalar value and *v* is a vector.

Scalar-Vector Vector-Vector Conversions		**Destination**			
		scalar	**v2**	**v3**	**v4**
Source	**scalar**	-	[s, s]	[s, s, s]	[s, s, s, s]
	v2	v.0	-	[v.0, v.1, 0]	[v.0, v.1, 0, 0]
	v3	v.0	[v.0, v.1]	-	[v.0, v.1, v.2, 0]
	v4	v.0	[v.0, v.1]	[v.0, v.1, v.2]	-

Vector immediate values are specified similarly to aggregate initialization, but are not necessary unless the values are different (scalars are spread automatically). Some examples are shown below.

```
.global .v3 .f32 V;
add.v3.f32 V, V, 1;
cross.v3.f32 V, V, {0, 1, 0};
```

6.6. Operand Costs

Operands from different state spaces will affect the speed of an operation. Registers are fastest, while global memory is slowest. Much of the delay to memory can be hidden in a number of ways. The first is to have multiple threads of execution so that the hardware can issue a memory operation and then switch to other execution. Another way to hide latency is to issue the load instructions as early as possible, as execution is

not blocked until the desired result is used in a subsequent (in time) instruction. The register in a store operation is available much more quickly. Table 9 gives estimates of the costs of using different kinds of memory.

Space	Time	Notes
Register	0	
Shared	0	
Constant	0	Amortized cost is low, first access is high
Local	> 100 clocks	
Parameter	0	
Immediate	0	
Global	> 100 clocks	
Texture	> 100 clocks	
Surface	> 100 clocks	

Table 9: Cost estimates for accessing state-spaces

Section 7. Instruction Set

7.1. Format and Semantics of Instruction Descriptions

This section describes each PTX instruction. In addition to the name and the format of the instruction, the semantics are described, followed by some examples that attempt to show several possible instantiations of the instruction.

7.2. PTX Instructions

PTX instructions generally have from zero to four operands, plus an optional guard predicate appearing after an '@' symbol to the left of the opcode:

```
@P    opcode;
@P    opcode A;
@P    opcode D, A;
@P    opcode D, A, B;
@P    opcode D, A, B, C;
```

For instructions that create a result value, the D operand is the destination operand, while A, B, and C are the source operands.

The setp instruction writes two destination registers. We use a '|' symbol to separate multiple destination registers.

```
      setp.s32.lt p|q, a, b;  // p = (a < b); q = !(a < b);
```

For some instructions the destination operand is optional. A "bit bucket" operand denoted with an underscore ('_') may be used in place of a destination register.

7.3. Predicated Execution

In PTX, predicate registers are virtual and have .pred as the type specifier. So, predicate registers can be declared as

```
      .reg .pred p, q, r
```

All instructions have an optional "guard predicate" which controls conditional execution of the instruction. The syntax to specify conditional execution is to prefix an instruction with "@[!]p", where p is a predicate variable, optionally negated. Instructions without a guard predicate are executed unconditionally.

Predicates are most commonly set as the result of a comparison performed by the SETP instruction.

As an example, consider the high-level code

```
          if (i < n)
              j = j + 1;
```

This can be written in PTX as

```
      setp.lt.s32 p, i, n;    // p = (i < n)
@p    add.s32 j, j, 1;        // if i < n, add 1 to j
```

To get a conditional branch or conditional function call, use a predicate to control the execution of the branch or call instructions. To implement the above example as a true conditional branch, the following PTX instruction sequence might be used:

```
      setp.lt.s32 p, i, n;    // compare i to n
@!p   bra L1;                 // if false, branch over
      add.s32 j, j, 1;
L1:       …
```

7.3.1. Comparisons

7.3.1.1. Integer and Bit-Size Comparisons

The signed integer comparisons are the traditional **eq** (equal), **ne** (not-equal), **lt** (less-than), **le** (less-than-or-equal), **gt** (greater-than), and **ge** (greater-than-or-equal). The unsigned comparisons are **eq**, **ne**, **lo** (lower), **ls** (lower-or-same), **hi** (higher), and **hs** (higher-or-same). The bit-size comparisons are **eq** and **ne**; ordering comparisons are not defined for bit-size types. The following table shows the operators for signed integer, unsigned integer, and bit-size types.

Meaning	Signed Operator	Unsigned Operator	Bit-Size Operator
a == b	EQ	EQ	EQ
a != b	NE	NE	NE
a < b	LT	LO	
a <= b	LE	LS	
a > b	GT	HI	
a >= b	GE	HS	

7.3.1.2. Floating-point Comparisons

The ordered comparisons are **eq**, **ne**, **lt**, **le**, **gt**, **ge**. If either operand is NaN, the result is **false**.

Meaning	Floating-Point Operator
a == b && !isNaN(a) && !isNaN(b)	EQ
a != b && !isNaN(a) && !isNaN(b)	NE
a < b && !isNaN(a) && !isNaN(b)	LT
a <= b && !isNaN(a) && !isNaN(b)	LE
a > b && !isNaN(a) && !isNaN(b)	GT
a >= b && !isNaN(a) && !isNaN(b)	GE

To aid comparison operations in the presence of NaN values, unordered versions are included: **equ**, **neu**, **ltu**, **leu**, **gtu**, **geu**. If both operands are numeric values (not NaN), then these comparisons have the same result as their ordered counterparts. If either operand is NaN, then the result of these comparisons is **true**.

Meaning	Floating-Point Operator
a == b || isNaN(a) || isNaN(b)	EQU
a != b || isNaN(a) || isNaN(b)	NEU
a < b || isNaN(a) || isNaN(b)	LTU
a <= b || isNaN(a) || isNaN(b)	LEU
a > b || isNaN(a) || isNaN(b)	GTU
a >= b || isNaN(a) || isNaN(b)	GEU

To test for NaN values, two operators **num** (numeric) and **nan** (isNaN) are provided. **num** returns **true** if both operands are numeric values (not NaN), and **nan** returns **true** if either operand is NaN.

Meaning	Floating-Point Operator
!isNaN(a) && !isNaN(b)	NUM
isNaN(a) || isNaN(b)	NAN

7.3.2. Manipulating Predicates

Predicate values may be computed and manipulated using the following instructions: **and**, **or**, **xor**, **not**, and **mov**.

There is no direct conversion between predicates and integer values, and no direct way to load or store predicate register values. However, **setp** can be used to generate a predicate from an integer, and the predicate-based select (**selp**) instruction can be used to generate an integer value based on the value of a predicate; for example:

```
selp.u32 %r1,1,0,%p;   // convert predicate to 32-bit value
```

7.4. Type Information for Instructions and Operands

Instructions that have a type must have a type suffix, e.g. **add.u16** or **add.f32**. The operand type must agree with the instruction type suffix. The bit-size types agree with any type of the same size. For example, the **add** instruction requires type and size information to properly perform the addition operation (signed, unsigned, float, different sizes), and this information must be specified as a suffix to the opcode.

Example:

```
add.u16 d, a, b;    // perform a 16-bit unsigned add
```

Integer types are compatible provided they have the same size, and integer operands are silently cast to the instruction type if needed. For example, an unsigned integer operand used in a signed integer instruction will be treated as a signed integer by the instruction.

Example:

```
.reg .u32 x;
.reg .s32 a;

neg.s32 a, x;       // signed negation of x
```

Some instructions require multiple type and size declarations, most notably the data conversion instruction **cvt**. It requires types for the result and source, and these are placed in the same order as the operands. For example:

```
cvt.f32.u16 d, a;   // convert 16-bit unsigned to 32-bit float
```

7.5. Divergence of Threads in Control Constructs

Threads in a CTA execute together, at least in appearance, until they come to a conditional control construct such as a conditional branch, conditional function call, or conditional return. If threads execute down different control flow paths, the threads are called *divergent*. If all of the threads act in unison and follow a single control flow path, the threads are called *uniform*. Both situations occur often in programs.

A CTA with divergent threads may have lower performance than a CTA with uniformly executing threads, so it is important to have divergent threads reconverge as soon as possible. All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the .**uni** suffix. For divergent control flow, the optimizing code generator automatically determines points of reconvergence. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent.

7.6. Semantics

The goal of the semantic description of an instruction is to describe the results in all cases in as simple language as possible. The semantics are described using C, until C is not expressive enough.

7.6.1. Machine-specific Semantics of 16-bit Code

A PTX program may execute on a GPU with either a 16-bit or a 32-bit datapath. When executing on a 32-bit datapath, 16-bit registers in PTX are mapped to 32-bit physical registers, and 16-bit computations are "promoted" to 32-bit computations. This can lead to computational differences between code run on a 16-bit machine versus the same code run on a 32-bit machine, since the "promoted" computation may have bits in the high-order half-word of registers that are not present in 16-bit physical registers. These extra precision bits can become visible at the application level, for example, by a right-shift instruction.

At the PTX language level, one solution would be to define semantics for 16-bit code that is consistent with execution on a 16-bit datapath. This approach introduces a performance penalty for 16-bit code executing on a 32-bit datapath, since the translated code would require many additional masking instructions to suppress extra precision bits in the high-order half-word of 32-bit registers.

Rather than introduce a performance penalty for 16-bit code running on 32-bit GPUs, the semantics of 16-bit instructions in PTX is machine-specific. A compiler or programmer may chose to enforce portable, machine-independent 16-bit semantics by adding explicit conversions to 16-bit values at appropriate points in the program to gurantee portability of the code. However, for many performance-critical applications, this is not desirable, and for many applications the difference in execution is preferable to limiting performance.

7.7. Instructions

All PTX instructions may be predicated. In the following descriptions, the optional guard predicate is omitted from the syntax.

7.7.1. Arithmetic Instructions

These instructions operate on the numeric types in register, vector, and constant immediate forms.

ADD	Add two values

Syntax	`add[.`*`rnd`*`][.sat].`*`type`* `d, a, b;`
	
	`.`*`type`* `= { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Performs addition and writes the resulting value into a destination register.
Semantics	`d = a + b;`
Integer Notes	No integer rounding modes. Saturation mode: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type.
Floating Point Notes	Rounding modes: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero Saturation mode: .sat limits result to (0.0, 1.0). Applies only to .f32 type.

Examples	`@p add.u32 x,y,z;`
	`add.sat.s32 c,c,1;`
	`add.rz.f32 f1,f2,f3;`

SUB	Subtract one value from another

Syntax	`sub[.`*`rnd`*`][.sat].`*`type`* `d, a, b;`
	
	`.`*`type`* `= { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Performs subtraction and writes the resulting value into a destination register.
Semantics	`d = a – b;`
Integer Notes	No integer rounding modes. Saturation mode: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type.
Floating Point Notes	Rounding modes: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero Saturation mode: .sat limits result to (0.0, 1.0). Applies only to .f32 type.
Examples	`sub.s32 c,a,b;`

MUL	Multiply two values

Syntax	`mul[.hi,.lo,.wide][.`*`rnd`*`][.sat].`*`type`* `d, a, b;`
	
	`.`*`type`* `= { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Compute the product of two values.

Semantics	t = a * b;
	n = bitwidth of type;
	d = t; // for floating-point and .wide
	d = t<2n-1..n>; // for .hi variant
	d = t<n-1..0>; // for .lo variant

Integer Notes	The type of the operation represents the types of the **a** and **b** operands. If .hi or .lo is specified, then **d** is the same size as **a** and **b**, and either the upper or lower half of the result is written to the destination register. If .wide is specified, then **d** is twice as wide as **a** and **b** to receive the full result of the multiplication. The .wide suffix is supported only for 16- and 32-bit integer types. No integer saturation.
Floating Point Notes	For floating-point multiplication, all operands must be the same size. Rounding modes: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero Saturation mode: .sat limits result to (0.0, 1.0). Applies only to .f32 type.

Examples	`mul.wide.s16 fa,fxs,fys; // 16*16 bits yields 32 bits`
	`mul.lo.s16 fa,fxs,fys; // 16*16 bits, save only the low 16 bits`
	`mul.wide.s32 z,x,y; // 32*32 bits, creates 64 bit result`
	`mul.f32 circumf,radius,pi // a single-precision multiply`

MAD	Multiply two values and add a third value

Syntax	`mad[.hi,.lo,.wide][.`*`rnd`*`][.sat].`*`type`* `d, a, b, c;`
	
	`.`*`type`* `= { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Multiplies two values and adds a third, and then writes the resulting value into a destination register.

Semantics	t = a * b;
	n = bitwidth of type;
	d = t + c; // for floating-point and .wide
	d = t<2n-1..n> + c; // for .hi variant
	d = t<n-1..0> + c; // for .lo variant

Integer Notes	The type of the operation represents the types of the **a** and **b** operands. If .hi or .lo is specified, then **d** and **c** are the same size as **a** and **b**, and either the upper or lower half of the result is written to the destination register. If .wide is specified, then **d** and **c** are twice as wide as **a** and **b** to receive the result of the multiplication. The .wide suffix is supported only for 16- and 32-bit integer types. Saturation mode: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s32 type in .hi or .lo mode.
Floating Point Notes	Rounding modes: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero Saturation mode: .sat limits result to (0.0, 1.0). Applies only to .f32 type.

Examples	`mad.lo.s32 d,a,b,c;`
	`mad.lo.s32 r,p,q,r;`
	`@p mad.f32 d,a,b,c;`

MUL24	Multiply two 24-bit integer values
Syntax	`mul24[.hi,.lo].`*`type`* `d, a, b;` *`.type`* `= { .u32, .s32 };`
Description	Compute the product of two 24-bit integer values held in 32-bit source registers, and return either the high or low 32-bits of the 48-bit result.

Semantics	t = a * b;
	d = t<47..16>; // for .hi variant
	d = t<31..0>; // for .lo variant

Notes	Integer multiplication yields a result that is twice the size of the input operands, i.e. 48-bits. mul24.hi performs a 24x24-bit multiply and returns the high 32 bits of the 48-bit result. mul24.lo performs a 24x24-bit multiply and returns the low 32 bits of the 48-bit result. All operands are of the same type and size. No saturation. mul24.hi may be less efficient on machines without hardware support for 24-bit multiply.
Examples	`mul24.lo.s32 d,a,b; // low 32-bits of 24x24-bit signed multiply.`

MAD24	Multiply two 24-bit integer values and add a third value
Syntax	`mad24[.hi,.lo][.sat].`*`type`* `d, a, b, c;` *`.type`* `= { .u32, .s32 };`
Description	Compute the product of two 24-bit integer values held in 32-bit source registers, and add a third, 32-bit value to either the high or low 32-bits of the 48-bit result. Return either the high or low 32-bits of the 48-bit result.

Semantics	t = a * b;
	d = t<47..16> + c; // for .hi variant
	d = t<31..0> + c; // for .lo variant

Notes	Integer multiplication yields a result that is twice the size of the input operands, i.e. 48-bits. mad24.hi performs a 24x24-bit multiply and adds the high 32 bits of the 48-bit result to a third value. mad24.lo performs a 24x24-bit multiply and adds the low 32 bits of the 48-bit result to a third value. All operands are of the same type and size. Saturation mode: .sat limits result of 32-bit signed addition to MININT..MAXINT (no overflow). Applies only to .s32 type. mad24.hi may be less efficient on machines without hardware support for 24-bit multiply.
Examples	`mad24.lo.s32 d,a,b,c; // low 32-bits of 24x24-bit signed multiply.`

SAD	Sum of absolute differences

Syntax	`sad[.rnd].type d, a, b, c;`
	
	`.type = { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Adds the absolute value of **a**-**b** to **c** and writes the resulting value into a destination register.
Semantics	`d = |a-b| + c;`
Floating Point Notes	Rounding modes: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero

Examples	`sad.s32 d,a,b,c;`
	`sad.u32 d,a,b,d; // running sum`
	`sad.f32 w,x,y,z;`

DIV	Divide one value by another

Syntax	`div[.wide][.`*`rnd`*`][.sat].`*`type`* `d, a, b;`
	
	`.`*`type`* `= { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Divides **a** by **b**, stores result in **d**.
Semantics	`d = a / b;`
Integer Notes	The .wide suffix specifies that **a** is twice the size of **b** and **d**. Otherwise, all three operands are the same size. The .wide suffix is supported only for 16- and 32-bit integer types. Division by zero yields an unspecified, machine-specific value. No integer saturation.
Floating Point Notes	Division by zero creates a value of infinity (with same sign as **a**). Rounding modes: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero Saturation mode: .sat limits result to (0.0, 1.0). Applies only to .f32 type.
Release Notes	`div.wide and div.{u64,s64} are not implemented in Release 1.0.`

Examples	`div.s32 b,n,i;`
	`div.wide.s32 d,an_s64_var,b;`
	`div.f32 diam,circum,3.14159;`

REM	The remainder of integer division

Syntax	`rem[.wide].`*`type`* `d, a, b;`
	
	`.type` `= { .u16, .u32, .u64,`
	`.s16, .s32, .s64 };`

Description	Divide **a** by **b**, store the remainder in **d**.
Semantics	`d = a % b;`
Integer Notes	The .wide suffix specifies that **a** is twice the size of **b** and **d**. Otherwise, all three operands are the same size. The .wide suffix is supported only for 16- and 32-bit integer types. The behavior for negative numbers is machine-dependent and depends on whether divide rounds towards zero or negative infinity.
Floating Point Notes	No floating-point support.
Release Notes	`rem.wide and rem.{u64,s64} are not implemented in Release 1.0.`
Examples	`rem.s32 x,x,8; // x = x%8;`

ABS	Absolute value

Syntax	`abs.`*`type`* `d, a;`
	
	`.type` `= { .s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Take the absolute value of **a** and store it in **d**.
Semantics	d = |a|;
Notes	Only for signed integers and floating-point numbers.
Examples	`abs.s32 r0,a;` `abs.f32 x,f0;`

NEG	Arithmetic negate

Syntax	`neg.`*`type`* `d, a;`
	
	`.type` `= { .s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Subtract **a** from zero and store the result in **d**.
Semantics	d = 0-a;
Notes	Only for signed integers and floating-point numbers.
Examples	`neg.s32 r0,a;` `neg.f32 x,f0;`

MIN	Find the minimum of two values

Syntax	`min.type d, a, b;`
	
	`.type = { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Store the minimum of **a** and **b** in **d**.
Semantics	`d = (a < b) ? a : b;` // Integer (signed and unsigned) `d = isNaN(a) ? b : isNan(b) ? a : (a < b) ? a : b;` // Floating Point
Integer Notes	Signed and unsigned differ.
Floating Point Notes	If either source operand is NaN, then the result is the other operand.

Examples	`min.s32 r0,a,b;`
	`@p min.u16 h,i,j;`
	`min.f32 z,z,x;`

MAX	Find the maximum of two values

Syntax	`max.type d, a, b;`
	
	`.type = { .u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Store the maximum of **a** and **b** in **d**.
Semantics	`d = (a > b) ? a : b;` // Integer (signed and unsigned) `d = isNan(a) ? b : isNan(b) ? a : (a > b) a : b;` // Floating Point
Integer Notes	Signed and unsigned differ.
Floating Point Notes	If either source operand is NaN, then the result is the other operand.

Examples	`max.f32 f0,f1,f2;`
	`max.u32 d,a,b;`
	`max.s32 q,q,0;`

7.7.2. Comparison and Selection Instructions

SET	Compare two numeric values with a relational operator, and optionally combine this result with a predicate value by applying a Boolean operator

Syntax	`set.`*`CmpOp.dtype.stype`* `d, a, b;`
	`set.`*`CmpOp.BoolOp.dtype.stype`* `d, a, b, [!]c;`
	
	`.dtype` `= { .u32, .s32, .f32 };`
	`.stype` `= { .b16, .b32, .b64,`
	`.u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Compares two numeric values and optionally combines the result with another predicate value by applying a Boolean operator. If this result is True, 1.0f is written for floating-point destination types, and 0xFFFFFFFF is written for integer destination types. Otherwise, 0x00000000 is written. The comparison operator is a suffix on the instruction, and can be one of: **eq**, **ne**, **lt**, **le**, **gt**, **ge** **lo**, **ls**, **hi**, **hs** **equ**, **neu**, **ltu**, **leu**, **gtu**, **geu** **num**, **nan** The Boolean operator BoolOp(A,B) is one of: **and**, **or**, **xor**

Semantics	t = (a *CmpOp* b) ? 1 : 0;
	if (isFloat(*dtype*)) { d = *BoolOp*(t, c) ? 1.0f : 0x00000000; }
	else { d = *BoolOp*(t, c) ? 0xFFFFFFFF : 0x00000000; }

Integer Notes	The signed comparisons are **eq**, **ne**, **lt**, **le**, **gt**, **ge**. The unsigned comparisons are **eq**, **ne**, **lo**, **hi**, **ls**, and **hs** for lower, higher, lower-or-same, and higher-or-same. The untyped, bit-size comparisons are **eq** and **ne.**
Floating Point Notes	The ordered comparisons are **eq**, **ne**, **lt**, **le**, **gt**, **ge**. If either operand is NaN, the result is **false**. To aid comparison operations in the presence of NaN values, unordered versions are included: **equ**, **neu**, **ltu**, **leu**, **gtu**, **geu**. If both operands are numeric values (not NaN), then these comparisons have the same result as their ordered counterparts. If either operand is NaN, then the result of these comparisons is **true.** **num** returns **true** if both operands are numeric values (not NaN), and **nan** returns **true** if either operand is NaN.
Examples	`set.lt.and.f32.s32 d,a,b,r;` `set.eq.u32.u32 d,i,n;`

SETP	Compare two numeric values with a relational operator, and (optionally) combine this result with a predicate value by applying a Boolean operator

Syntax	`setp.`*`CmpOp.type`* `p[|q], a, b;`
	`setp.`*`CmpOp.BoolOp.type`* `p[|q], a, b, [!]c;`
	
	`.`*`type`* `= { .b16, .b32, .b64,`
	`.u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Compares two values and combines the result with another predicate value by applying a Boolean operator. This result is written to the first destination operand. A related value computed using the complement of the compare result is written to the second destination operand. Applies to all numeric types. The comparison operator is a suffix on the instruction, and can be one of: **eq**, **ne**, **lt**, **le**, **gt**, **ge** **lo**, **ls**, **hi**, **hs** **equ**, **neu**, **ltu**, **leu**, **gtu**, **geu** **num**, **nan** The Boolean operator BoolOp(A,B) is one of: **and**, **or**, **xor** The destinations p and q must be .pred variables.

Semantics	t = (a CmpOp b) ? 1 : 0;
	p = BoolOp(t, c);
	q = BoolOp(!t, c);

Integer Notes	The signed comparisons are **eq**, **ne**, **lt**, **le**, **gt**, **ge**. The unsigned comparisons are **eq**, **ne**, **lo**, **hi**, **ls**, and **hs** for lower, higher, lower-or-same, and higher-or-same. The untyped, bit-size comparisons are **eq** and **ne.**
Floating Point Notes	The ordered comparisons are **eq**, **ne**, **lt**, **le**, **gt**, **ge**. If either operand is NaN, the result is **false**. To aid comparison operations in the presence of NaN values, unordered versions are included: **equ**, **neu**, **ltu**, **leu**, **gtu**, **geu**. If both operands are numeric values (not NaN), then these comparisons have the same result as their ordered counterparts. If either operand is NaN, then the result of these comparisons is **true**. **num** returns **true** if both operands are numeric values (not NaN), and **nan** returns **true** if either operand is NaN.
Examples	`setp.lt.and.s32 p|q,a,b,r;` `setp.eq.u32 p,i,n;`

SELP	Select between source operands, based on the value of the predicate source operand

Syntax	`selp.type d, a, b, c;`
	
	`.type = { .b16, .b32, .b64,`
	`.u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Conditional selection. If **c** is True, **a** is stored in **d**, **b** otherwise. Operands **d**, **a**, and **b** must be of the same type. Operand **c** is a predicate.
Semantics	`d = (c == 1) ? a : b;`
Examples	`selp.s32 r0,r,g,p;` `selp.f32 f0,t,x,xp;`

SLCT	Select one source operand, based on the sign of the third operand

Syntax	`slct.dtype.ctype d, a, b, c;`
	
	`.dtype = { .b16, .b32, .b64,`
	`.u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`
	`.ctype = { .s32, .f32 };`

Description	Conditional selection. If **c**>=0, **a** is stored in **d**, **b** otherwise. Operands **d**, **a**, and **b** are treated as a bitsize type of the same width as the first instruction type; operand **c** must match the second instruction type.
Semantics	`d = (c >= 0) ? a : b;` For .f32 comparisons, if operand **c** is a denorm, it is flushed to zero, resulting in selection of operand **a**. If operand **c** is NaN, the comparison is unordered and operand **b** is selected.
Floating Point Notes	For .f32 data selections, denorm results are flushed to zero.
Examples	`slct.u32.s32 x, y, z, val;` `slct.u64.f32 A, B, C, fval;`

7.7.3. Logic and Shift Instructions

The logic and shift instructions are fundamentally untyped, performing bit-wise operations on operands of any type, provided the operands are of the same size. This permits bit-wise operations on floating point values without having to define a union to access the bits. Instructions **and**, **or**, **xor**, and **not** also operate on predicates.

AND	Bitwise AND
Syntax	`and.`*`type`* `d, a, b;` *`type`* `= { .pred, .b16, .b32, .b64 };`
Description	Compute the bit-wise and operation for the bits in **a** and **b**.
Semantics	`d = a & b;`
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
Examples	`and.b32  x,q,r;` `and.b32  sign,fpvalue,0x80000000;`

OR	Bitwise OR
Syntax	`or.`*`type`* `d, a, b;` *`type`* `= { .pred, .b16, .b32, .b64 };`
Description	Compute the bit-wise or operation for the bits in **a** and **b**.
Semantics	`d = a | b;`
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
Examples	`or.b32  mask mask,0x00010001` `or.pred  p,q,r;`

XOR	Bitwise exclusive-or (inequality)
Syntax	`xor.`*`type`* `d, a, b;` *`.type`* `= { .pred, .b16, .b32, .b64 };`
Description	Compute the bit-wise exclusive-or of the bits in **a** and **b**.
Semantics	`d = a ^ b;`
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicate registers.
Examples	`xor.b32  d,q,r;` `xor.b16  d,x,0x0001;`

NOT	Bitwise negation; one's complement
Syntax	`not.`*`type`* `d, a;` *`.type`* `= { .pred, .b16, .b32, .b64 };`
Description	Invert the bits in **a**.
Semantics	`d = ~a;`
Notes	The size of the operands must match, but not necessarily the type. Allowed types include predicates.
Examples	`not.b32  mask,mask;` `not.pred  p,q;`

CNOT	C/C++ style logical negation
Syntax	`cnot.`*`type`* `d, a;` *`.type`* `= { .b16, .b32, .b64 };`
Description	Compute the logical negation using C/C++ semantics.
Semantics	`d = (a==0) ? 1 : 0;`
Notes	The size of the operands must match, but not necessarily the type.
Examples	`cnot.b32 d,a;`

SHL	Shift bits left, zero-fill on right
Syntax	`shl.`*`type`* `d, a, b;` *`.type`* `= { .b16, .b32, .b64 };`
Description	Shift **a** left by the amount specified by **b**.
Semantics	`d = a << b;`
Notes	Shift amounts greater than the register width *N* are clamped to *N*. The size of the operands must match, but not necessarily the type.
Examples	`shl.b32  q,a,2;`

SHR	Shift bits right, sign or zero fill on left

Syntax	`shr.`*`type`* `d, a, b;`
	
	`.type` `= { .b16, .b32, .b64,`
	`.u16, .u32, .u64,`
	`.s16, .s32, .s64 };`

Description	Shift **a** right by the amount specified by **b**. Signed shifts fill with the sign bit, unsigned and untyped shifts fill with 0.
Semantics	`d = a >> b;`
Notes	Shift amounts greater than the register width *N* are clamped to *N*. Bit-size types are included for symmetry with SHL.

Examples	`shr.u16  c,a,2;`
	`shr.s32  i,i,1;`
	`shr.b16  k,i,j;`

7.7.4. Data Movement and Conversion Instructions

These instructions copy data from place to place, and from state space to state space, possibly converting it from one format to another.

MOV	Set a register variable with the value of a register variable or an immediate value

Syntax	`mov.type d, a;`
	`mov.type d, sreg;  // sizes must match`
	`mov.type d, avar;  // move address of variable into destination reg`
	
	`.type = { .pred,`
	`.b16, .b32, .b64,`
	`.u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32, .f64 };`

Description	Write register **d** with the value of **a**. Operand **a** may be a register, special register, immediate, or addressable variable.
Semantics	`d = a;`
Notes	Although only predicate and bit-size types are required, we include the arithmetic types for the programmer's convenience: their use enhances program readability and allows additional type checking.

Examples	`mov.f32  d,a;`
	`mov.u16  u,v;`
	`mov.f32  k,0.1;`
	`mov.u32  ptr, A;  // move address of A into ptr`
	`mov.u32  ptr, A[5]; // move address of A[5] into ptr`

LD	Load a register variable from an addressable state space variable

Syntax	ld.*space*.*type* d,[a]; // load from address
	ld.*space*.*vec*.*type* d,[a]; // vector load from address
	
	.*space* = { .const, .global, .local, .param, .shared };
	.*vec* = { .v2, .v3, .v4 };
	.*type* = { .b8, .b16, .b32, .b64,
	.u8, .u16, .u32, .u64,
	.s8, .s16, .s32, .s64,
	.f32, .f64 };

Description	Load register variable **d** from the location specified by the source address operand **a**. The addressable operand **a** is one of: [*avar*] the name of an addressable variable *var*, [*areg*] a register *reg* containing a byte address, [*areg+immOff*] a sum of register *reg* containing a byte address plus a constant integer byte offset (signed, 32-bit), or [*immAddr*] an immediate absolute byte address (unsigned, 32-bit). The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. The instruction must carry a .*space* suffix. A register containing an address may be declared as a bit-size type or integer type.

Semantics	d = a; // named variable a
	d = *a; // register
	d = *(a+immOff); // register-plus-offset
	d = *(immAddr); // immediate address

Notes	Destination **d** must be in the .reg state space. For integer loads, if the destination register is wider than the specified type, the value loaded is extended to the destination register width. The type of extension (sign or zero) is determined by the .*type* field. .f16 data may be loaded using **ld.b16**, and then converted to .f32 or .f64 using **cvt**.
Examples	ld.global.f32 d,[a]; ld.shared.b32 d,[p]; ld.const.s32 d,[p+4]; ld.global.v4.f32 Q,[p]; ld.local.b64 x,[240]; // load from location 240 in space local

ST	Store a register variable to an addressable state space variable

Syntax	`st.`*`space`*`.`*`type`* [d],a; // store to address
	`st.`*`space`*`.`*`vec`*`.`*`type`* [d],a; // vector store to address
	
	`.space` = {.global, .local, .shared };
	`.vec` = { .v2, .v3, .v4 };
	`.type` = { .b8, .b16, .b32, .b64,
	.u8, .u16, .u32, .u64,
	.s8, .s16, .s32, .s64,
	.f32, .f64 };

Description	Store the value of register variable **a** in the location specified by the destination address operand **d**. The addressable operand **d** is one of: [*var*] the name of an addressable variable *var*, [*reg*] a register *reg* containing a byte address, [*reg+immOff*] a sum of register *reg* containing a byte address plus a constant integer byte offset (signed, 32-bit), or [*immAddr*] an immediate absolute byte address (unsigned, 32-bit). The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. The instruction must carry a *.space* suffix. A register containing an address may be declared as a bit-size type or integer type.

Semantics	d = a; // named variable *d*
	*d = a; // register
	*(d+immOffset) = a; // register-plus-offset
	*(immAddr) = a; // immediate address

Notes	Operand **a** must be in the .reg state space. .f16 data resulting from a **`cvt`** instruction may be stored using **`st.b16`**.

Examples	`st.global.f32 [d],a;`
	`st.local.b32 [q+4],a;`
	`st.global.v4.s32 [p],Q;`
	`st.shared.s32 [100],r7;`

CVT	Convert a value from one type to another

Syntax	`cvt[.`*`rnd`*`][.sat].`*`dtype`*`.`*`atype`* `d, a;`
	
	`.dtype` `= .`*`atype`* `= { .u8, .u16, .u32, .u64,`
	`.s8, .s16, .s32, .s64,`
	`.f16, .f32, .f64 };`

Description	Convert between different types and sizes. See the Integer and Floating-point Notes below for details of rounding modes.
Semantics	d = convert(a);
Integer Notes	Integer rounding modes: .rni round to nearest integer, choosing even integer if source is equidistance between two integers. .rzi round to nearest integer in the direction of zero .rmi round to nearest integer in direction of negative infinity .rpi round to nearest integer in direction of positive infinity Saturation mode: .sat limits result to MININT..MAXINT (no overflow) for the size of the operation. Applies only to .s16, .s32, and .s64 types.
Floating Point Notes	If the source and destination are both floating-point, then the rounding modes describe how to set the lsb's when there is a loss of precision. Floating-point rounding modes: .rn mantissa LSB rounds to nearest even .rz mantissa LSB rounds towards zero .rm mantissa LSB rounds towards negative infinity .rp mantissa LSB rounds towards positive infinity A floating-point value may be rounded to an integral value using the integer rounding modes (see Integer Notes). The operands must be of the same size. The result is an integral value, stored in floating-point format. Saturation mode: .sat limits result to (0.0, 1.0). Applies to .f16, .f32, and .f64 types. NaN is preserved, except for .f16 (no NaN available).

Examples	`cvt.f32.s32 f,i;`
	`cvt.sat.s32.f64 j,r;`
	`cvt.rni.f32.f32 x,y; // round fp val to nearest int, result is fp`

7.7.5. Texture Instruction

TEX	Perform a texture memory lookup

Syntax	`tex.`*`geom`*`.`*`dtype`*`.`*`btype`* `d, a, b;`
	
	`.geom` `= { .1d, .2d, .3d };`
	`.dtype` `=`
	`.btype` `= { .b16, .b32, .b64,`
	`.u16, .u32, .u64,`
	`.s16, .s32, .s64,`
	`.f32 };`

Description	Texture lookup using a texture coordinate vector.
Examples	`tex.3d.v4.s32.f32 {r1,r2,r3,r4},tex_a,{f1,f2,f3};` `tex.1d.v4.s32.f32 {r1,r2,r3,r4},tex_a,{f1};`

7.7.6. Control Flow Instructions

These PTX instructions and syntax are for controlling execution in a PTX program.

{ }	Instruction grouping
Syntax	`{ instructionList }`
Description	The curly braces create a group of instructions, used primarily for defining a function body. The curly braces also provide a mechanism for determining the scope of a variable: any variable declared within a scope is not available outside the scope.
Examples	`{ add.s32 a,b,c; mov.s32 d,a; }`

@	Predicated execution
Syntax	`@[!]p instruction;`
Description	Execute an instruction or instruction block for threads that have the guard predicate true. Threads with a false guard predicate do nothing.
Semantics	If [!]p then *instruction*

Examples	`setp.eq.f32 p,y,0; // is y zero?`
	`@!p div.f32 ratio,x,y // avoid division by zero`
	
	`@q bra L23; // conditional branch`

BRA	Branch to a target and continue execution there
Syntax	`bra[.uni] target;`
Description	Continue execution at the target. Conditional branches are specified with the '@' prefix.
Semantics	pc = target;
Notes	A bra is assumed to be divergent unless the .uni suffix is present, indicating that the branch is guaranteed to be non-divergent.
Release Notes	`Indirect branch through a register is not supported in Release 1.0.`
Examples	`bra.uni L_exit;` `@p bra L321;`

CALL	Call a function, recording the return location

Syntax	`call[.uni] fname;`
	`call[.uni] fname, (param-list);`
	`call[.uni] (ret-param), fname, (param-list);`

Description	Call a function, storing current execution information for subsequent return.
Notes	The call instruction stores the address of the next instruction, so execution can resume at that point after executing a RET instruction. The called location can be either a symbolic function name or an address held in a register. A call is assumed to be divergent unless the .uni suffix is present, indicating that the call is guaranteed to be non-divergent. Input and return parameters are optional. Parameters must be of register type, and parameters are pass-by-value. In the current ptx release, parameters are passed through statically allocated ptx registers; i.e., there is no support for recursive calls..

Examples	`call init; // call function 'init'`
	`call.uni g, (a); // call function 'g' with parameter 'a'`
	`@p call (d), h, (a, b); // return value into register d`

RET	Return from function to instruction after call
Syntax	`ret[.uni];`
Description	Return execution to caller's environment. A divergent return suspends threads until all threads are ready to return to the caller. This allows multiple divergent "ret" instructions.
Notes	A ret is assumed to be divergent unless the .uni suffix is present, indicating that the return is guaranteed to be non-divergent. Any values returned from a function should be moved into the return parameter register variables prior to executing the RET instruction. A return instruction executed in a top-level entry routine will terminate thread execution.
Examples	`    ret;` `@p  ret;`

EXIT	Terminate a thread
Syntax	`exit;`
Description	Ends execution of a thread.
Examples	`    exit;` `@p  exit;`

7.7.7. Parallel Synchronization and Communication Instructions

BAR	Signal arrival at a barrier,and wait
Syntax	`bar.sync d;`
Description	Marks the arrival of threads at a barrier and waits for all other threads to arrive. The barrier resource is named via a small integer, typically in the range 0..15. The barrier number may be given as an immediate.
Notes	The hardware has a limited, implementation-specific number of barrier resources, typically sixteen or fewer. Since a CTA will not launch until all allocated resources are available, a program should minimize the number of distinct barrier variables allocated. Ideally, a program uses a single, global barrier that is re-used throughout the program.
Examples	`bar.sync 0;`

ATOM	Atomic reduction operations for thread-to-thread communication

Syntax	`atom.space.operation.type d, a, b[, c];`
	
	`.space = { .global };`
	
	`.operation = { .and, .or, .xor,  // .b32 only`
	`  .inc, .dec,  // .u32 only`
	`  .add,  // .u32, .s32, .f32,`
	`  .min, .max,  // .u32, .s32, .f32`
	`  .cas, .exch };  // all types`
	
	`.type = { .b32, .u32, .s32, .f32 };`

Description	Atomically loads the original value at location **a** into destination register **d**, and stores the result of the specified operation at location **a**, overwriting the original value. The **a** operand specifies a location in the specified state space. The addressable operand **a** is one of: [avar] the name of an addressable variable *avar*, [areg] a de-referenced register *areg* containing a byte address, [areg+immOff] a de-referenced sum of register *areg* containing a byte address plus a constant integer byte offset, or [immAddr] an immediate absolute byte address. The address size may be either 32-bit or 64-bit. Addresses are zero-extended to the specified width as needed, and truncated if the register width exceeds the state space address width for the target architecture. The instruction must carry a *.space* suffix. A register containing an address may be declared as a bit-size type or integer type. The bitsize operations are **and**, **or**, and **xor.** The integer operations are **cas** (compare-and-swap), **exch** (exchange), **add**, **inc**, **dec**, **min**, **max**. The **inc** and **dec** operations return a result in the range [0..**b**]. The floating-point operations are **add**, **min**, and **max**. The floating-point **add**, **min**, and **max** operations are 32-bit operations.

Semantics	atomic {
	  d = *a;
	  a = operation(*a, b);
	}
	
	where
	  inc(r, s) = (r >= s) ? 0 : r+1;
	  dec(r, s) = (r > s) ? s : r-1;

Notes	Operand **a** must reside in the global state space. Simple reductions may be specified by using the "bit bucket" destination operand '_'.

Target ISA Notes	`atom requires compute_11 or sm_11.`
Examples	`atom.global.add.s32  d,[a],1;`

7.7.8. Floating-point Instructions

These instructions are for floating-point types in register, vector, and constant immediate forms.

FRC	Save only the fractional part of a floating point value
Syntax	`frc.`*`type`* `d, a;` *`.type`* `= { .f32, .f64 };`
Description	Keep the fractional part of a floating-point value.
Semantics	d = a – floor(a);
Notes	Be careful with negative input values.
Examples	`frc.f32 f,g;`

SIN	Find the sine of a value

Syntax	`sin.`*`type`* `d, a;`
	
	`.`*`type`* `= { .f32, .f64 };`

Description	Find the sine of the angle **a** (in radians).
Semantics	d = sin(a);
Examples	`sin.f32  sa,a;`

COS	Find the cosine of a value

Syntax	`cos.`*`type`* `d, a;`
	
	`.`*`type`* `= { .f32, .f64 };`

Description	Find the cosine of the angle **a** (in radians).
Semantics	d = cos(a);
Examples	`cos.f32  cb,b;`

LG2	Find the log, base 2, of a value
Syntax	`lg2.`*`type`* `d, a;` *`.type`* `= { .f32, .f64 };`
Description	Determine the $\log_2$ of **a**.
Semantics	d = log(a)/log(2);
Floating Point Notes	If a < 0, d = NaN; If a == 0, d = -Inf;
Examples	`@p  lg2.f32  q,a;`

EX2	Exponentiate a value, base 2
Syntax	`ex2.`*`type`* `d, a;` *`.type`* `= { .f32, .f64 };`
Description	Raise 2 to the power **a**.
Semantics	d = 2 ^ a;
Floating Point Notes	If a == -Inf, d = 0; If a == Inf, d = Inf; If isNan(a), d = NaN;
Examples	`ex2.f32  q,r;`

RCP	Take the reciprocal of a value
Syntax	`rcp.`*`type`* `d, a;` *`.type`* `= { .f32, .f64 };`
Description	Compute 1/**a**.
Semantics	`d = 1/a;`
Floating Point Notes	1/0.0 yields +Inf. 1/NaN yields NaN
Examples	`rcp.f32 ri,r;`

SQRT	Take the square root of a value
Syntax	`sqrt.`*`type`* `d, a;` *`.type`* `= { .f32, .f64 };`
Description	Compute sqrt(**a**); store in **d**.
Semantics	d = sqrt(a);
Floating Point Notes	If a < 0; d = NaN; The sqrt instruction always yields the positive root of a number.
Examples	`sqrt.f32  r,x;`

RSQRT	Take the reciprocal of the square root of a value
Syntax	`rsqrt.`*`type`* `d, a;` *`.type`* `= { .f32, .f64 };`
Description	Compute 1/sqrt(**a**); store the result in **d**.
Semantics	d = 1/sqrt(a);
Floating Point Notes	If a < 0, d = NaN; If a == 0, d = Inf; The rsqrt instruction always yields a positive value.
Examples	`rsqrt.f32  isr,x;`

7.7.9. Miscellaneous Instructions

TRAP	Perform trap operation
Syntax	`trap;`
Description	Abort execution and generate an interrupt to the host CPU.
Examples	`trap;` `@p trap;`

BRKPT	Breakpoint – suspend execution
Syntax	`brkpt;`
Description	Suspends execution.
Target ISA Notes	`Unsupported in Release 1.0`
Examples	`brkpt;` `@p brkpt;`

Section 8. Special Registers

PTX includes a number of predefined, read-only variables, which are visible as special registers and accessed through MOV or CVT instructions.

%tid	Thread ID within a CTA

Syntax	.sreg .v3 .u16 %tid; // thread id vector
	.sreg .u16 %tid.0, %tid.1, %tid.2; // individual thread id components
	.sreg .u16 %tid.x, %tid.y, %tid.z; // alternate component names

Description	A predefined, read-only, per-thread special register initialized with the thread ID within the CTA. The %tid special register is a 1D, 2D, or 3D vector to match the CTA shape; the %tid value in unused dimensions is 0. The number of threads in each dimension are specified by the predefined special register %ntid. Every thread in the CTA has a unique %tid. %tid component values range from 0 through %ntid–1 in each CTA dimension. %tid.1 == %tid.2 == 0 in 1D CTAs. %tid.2 == 0 in 2D CTAs. It is guaranteed that: 0 <= %tid.0 < %ntid.0 0 <= %tid.1 < %ntid.1 0 <= %tid.2 < %ntid.2
Notes	3D CTA initialization code (or TID extraction code, which?) separates hardware %tid R0 bit fields [15:0, 25:16, 31:26] into 3 .u16 components in R0L, R0H, and R1L, and %tid maps to [R0L, R0H, R1L] in half words. 2D and 1D CTAs require no %tid initialization code. Preserve %tid for debugging.

Examples	`mov.b16 r0,%tid.0; // zero-extends tid.0 to r0`
	`cvt.u32.u16 r2,%tid.2; // zero-extends tid.2 to r2`

%ntid	Number of thread IDs per CTA

Syntax	.sreg .v3 .u16 %ntid; // CTA shape vector
	.sreg .u16 %ntid.0, %ntid.1, %ntid.2; // CTA dimensions
	.sreg .u16 %ntid.x, %ntid.y, %ntid.z; // alternate component names

Description	A predefined, read-only special register initialized with the number of thread ids in each CTA dimension. CTA dimensions are non-zero. The total number of threads in a CTA is (%ntid.0 * %ntid.1 * %ntid.2). The CTA dimensions are initialized in the predefined variable %ntid. The value of each element of the vector is at least 1. %ntid.1 == %ntid.2 == 1 in 1D CTAs. %ntid.2 == 1 in 2D CTAs.
Notes	

Examples	`mov.b16 r0,%tid.0;`
	`mov.b16 h1,%tid.1;`
	`mov.u16 h2,%ntid.0;`
	`mad.u16 r0,h1,h2,r0; // r0 = unified tid for 2D CTA`

%ctaid	CTA id within a grid

Syntax	.sreg .v3 .u16 %ctaid; // CTA id vector
	.sreg .u16 %ctaid.0, %ctaid.1, %ctaid.2; // CTA id components
	.sreg .u16 %ctaid.x, %ctaid.y, %ctaid.z; // alternate component names

Description	A predefined, read-only special register initialized with the CTA id within the CTA grid. %ctaid is a 1D, 2D, or 3D vector, depending on the shape and rank of the CTA grid. The value of each element of the vector is >= 0 and < 65535. It is guaranteed that: 0 <= %ctaid.0 < %nctaid.0 0 <= %ctaid.1 < %nctaid.1 0 <= %ctaid.2 < %nctaid.2
Notes	The G80 translator maps ctaid.0 to grid parameters g[6].u16, ctaid.1 to g[7].u16, and ctaid.2 to user parameter g[8].u16.
Examples	`mov.u32 %r1,%ctaid.1;`

%nctaid	Number of CTA ids per grid

Syntax	.sreg .v3 .u16 %nctaid; // Grid shape vector
	.sreg .u16 %nctaid.0, %nctaid.1, %nctaid.2; // Grid dimensions
	.sreg .u16 %nctaid.x, %nctaid.y, %nctaid.z; // alternate component names

Description	A predefined, read-only special register initialized with the number of CTAs in each grid dimension. %nctaid is a 1D, 2D, or 3D vector, depending on the shape and rank of the CTA grid. The size of the grid of CTAs is stored in the predefined special register %nctaid. It is a 3D vector, and each member has a value of at least 1. It is guaranteed that: 1 <= nctaid.* < 65,536
Notes	The G80 translator maps nctaid.0 to grid parameters g[4].u16, nctaid.1 to g[5].u16, and nctaid.2 to user parameter g[9].u16
Examples	`mov.u32 r1,%nctaid;`

%gridid	Grid ID
Syntax	.sreg .u16 %gridid; // initialized when the grid is launched
Description	A predefined, read-only special register initialized with the per-grid temporal grid ID number. This is used by debuggers to distinguish CTAs within concurrent (small) CTA grids. During execution, repeated launches of programs may occur, where each launch starts a *grid-of-CTAs*. This variable provides the temporal grid launch number for this context.
Notes	The driver assigns a counting sequential gridid to each grid launched. The G80 translator maps gridid to grid parameter g[0].u16, "flags".
Examples	`mov.u32 r1,%gridid;`

%clock	A predefined, read-only 32-bit unsigned cycle counter
Syntax	
Description	Special register %**clock** is an unsigned 32-bit read-only cycle counter that wraps silently.
Notes	
Examples	`mov.u32  r1,%clock;`

Section 9. Directives

9.1. Specifying CTAs and Functions

Directives exist for specifying CTA entry points, the default number of threads in a CTA, functions, and other things. Some can be overridden later on.

.entry	Defines a CTA entry point name, CTA rank, and optional CTA dimensions.
Syntax	**.entry** name **[***NTID.2***][***NTID.1***][***NTID.0***];** *// 3D CTA: RANKTID = 3.* **.entry** name **[***NTID.1***][***NTID.0***];** *// 2D CTA: RANKTID = 2.* **.entry** name **[***NTID.0***];** *// 1D CTA: RANKTID = 1.* **.entry** name **[***NTID.2***][***NTID.1***][***NTID.0***] {** *per-CTA naming scope* **}**
Description	Specifies a CTA entry point and name. The number of bracket pairs specifies the CTA rank constant RANKTID to be 1, 2, or 3. Constant expressions within the bracket pairs define the CTA dimension constants NTID.0, NTID.1, and NITD.2. Omitted dimension values define their constant dimension as 0. Omitted bracket pairs define their constant dimension as 1. Empty bracket pairs have unspecified dimensions that vary at run time, as specified by ntid.0, ntid.1, or ntid.2. Optionally specify a per-CTA naming scope enclosed in { } braces, for .shared and .param variable declarations. PTX appends an exit instruction following the code in the braces. PTX defines an anonymous 3D CTA entry point at the first instruction encountered outside of a .entry or .func block. PTX appends an exit instruction after the last instruction of an anonymous entry point.
Semantics	Specify the entry point for a CTA program. Defines the constants RANKTID, NTID.0, NTID.1, and NTID.2. At run time, the CTA parameters ntid.0, ntid.1, and ntid.2 are initialized with the actual CTA dimensions. The programmer may use the constant dimensions rather than the runtime dimensions if the CTA is always invoked with the constant dimensions.
Notes	CTA dimensions are positive integers; zero means a dimension is unknown until runtime. G80 limits the product of dimensions to 512.

Examples	`.entry cta_fft[256];  // entry of 1D CTA with max 256 threads.`
	`.entry filter[16][16] {  code; … }  // entry and scope for 2D CTA`

.func	Function definition.
Syntax	**.func** fname *function-body* **.func** fname (*param-list*) *function-body* **.func** (*ret-param*) fname (*param-list*) *function-body*
Description	Defines a function, including input and return parameters and function body.
Semantics	Specifies the entry point and parameter names for a function. The parameter lists bind register names in the caller's namespace to register names in the callee namespace. The implementation of parameter passing is left to the optimizing translator, which may use a combination of registers and stack locations to pass parameters. In the current ptx release, parameters are passed through statically allocated ptx registers; i.e., there is no support for recursive calls.
Notes	The input and return parameters are enclosed in parentheses. Parameters must be base types in the register space. Parameter passing is call-by-value. A .func directive with no body may be used to declare a function prototype.

Examples	`.func (.reg .b32 rval) foo (.reg .b32 arg0, .reg .f64 arg1)`
	`{`
	`.reg .b32 localVar;`
	
	`… use arg0;`
	`other code;`
	
	`mov.b32 rval,result;`
	`ret;`
	`}`
	
	`…`
	`call (fooval), foo, (val0, val1);  // return value in fooval`
	`…`

9.2. Debugging Directives

The following directives are needed to communicate Dwarf-format debug information. Details TBD.

.section	PTX section definition
Syntax	**.section** *section_type, section_name, ???*
Description	
Semantics	
Notes	
Examples	`.section .debug_info, "",@progbits`

.file	Source file information
Syntax	**.file** *filename*
Description	
Semantics	
Notes	
Examples	

.loc	Source file location
Syntax	**.loc** *line_number*
Description	
Semantics	
Notes	
Examples	

.byte	Byte data
Syntax	**.byte** *data-list*
Description	Defines a sequence of data bytes.
Semantics	
Notes	
Examples	`.byte  0x7d,0x01,0x00,0x00,0x02,0x00`

9.3. Other Directives

.extern	External symbol declaration
Syntax	**.extern** *identifier*
Description	Declares identifier to be defined externally.
Semantics	
Notes	
Examples	`.extern foo  // variable foo is declared in another file` `.b32 foo;`

.visible	Visible (externally) symbol declaration
Syntax	**.visible** *identifier*
Description	Declares identifier to be externally visible.
Semantics	
Notes	
Examples	`.visible foo  // variable foo will be externally visible` `.b32 foo;`

.version	PTX version number
Syntax	**.version** *major.minor // major, minor are integers*
Description	Specifies the PTX language version number. Increments to the major number indicate incompatible changes to PTX.
Semantics	Indicates that this file must be compiled with tools having the same major version number and an equal or greater minor version number. Each ptx file must begin with a .version directive. Duplicate .version directives are allowed provided they match the original .version directive.
Notes	Cuda Release 1.0 supports PTX ISA Version 1.0.
Examples	`.version 1.0`

.target	Architecture and Platform target
Syntax	**.target** *stringlist // comma separated list of target specifiers* *string* = { compute_10, compute_11, // virtual target architectures sm_10, sm_11,, // gpu target architectures map_f64_to_f32 // platform option };
Description	Specifies the target architecture for which the current ptx code was generated. The target identifier strings are platform-specific.
Semantics	PTX features are checked against the specified target architecture, and an error is generated if an unsupported feature is used. The map_f64_to_f32 specifier indicates that all double-precision instructions will be mapped to single-precision regardless of the target architecture. This feature enables compilers for high-level languages such as Cuda to compile programs containing type **double**. Each PTX file must begin with a .version directive, immediately followed by a .target directive. Duplicate .target directives are allowed provided they match the original .target directive.
Notes	

Examples	`.target sm_10  // baseline target architecture`
	
	`// allow .f64 instructions, but map them to .f32 in the translator`
	`.target sm_10, map_f64_to_f32  (required for CUDA Release 1.0)`

Section 10. Release 1.0 Notes

In Release 1.0 of the PTX ISA Version 1.0, a number of features are not supported. This section summarizes the unsupported features.

Syntax restrictions

Predicate constant immediates are not supported.

Constant expressions are not supported.

State Spaces

Declarations and instructions using .surf space are not supported.

The constant space is restricted to a single bank. This may be written as .const or .const[0].

Variables and Operands

Vector declarations, initialization, and conversions are not supported.

Vector operands are not generally supported. The LD, ST, and TEX instructions do support limited use of vector operands written using the tuple notation.

Instructions

See individual instruction descriptions in Section 7 for restrictions of the current release.