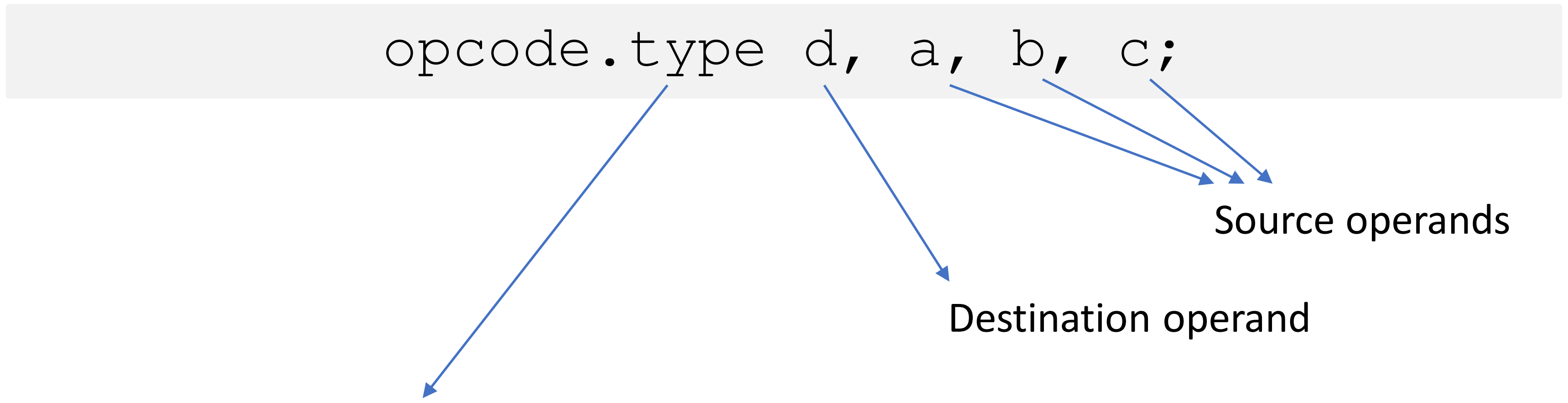
PTX: Parallel Thread Execution

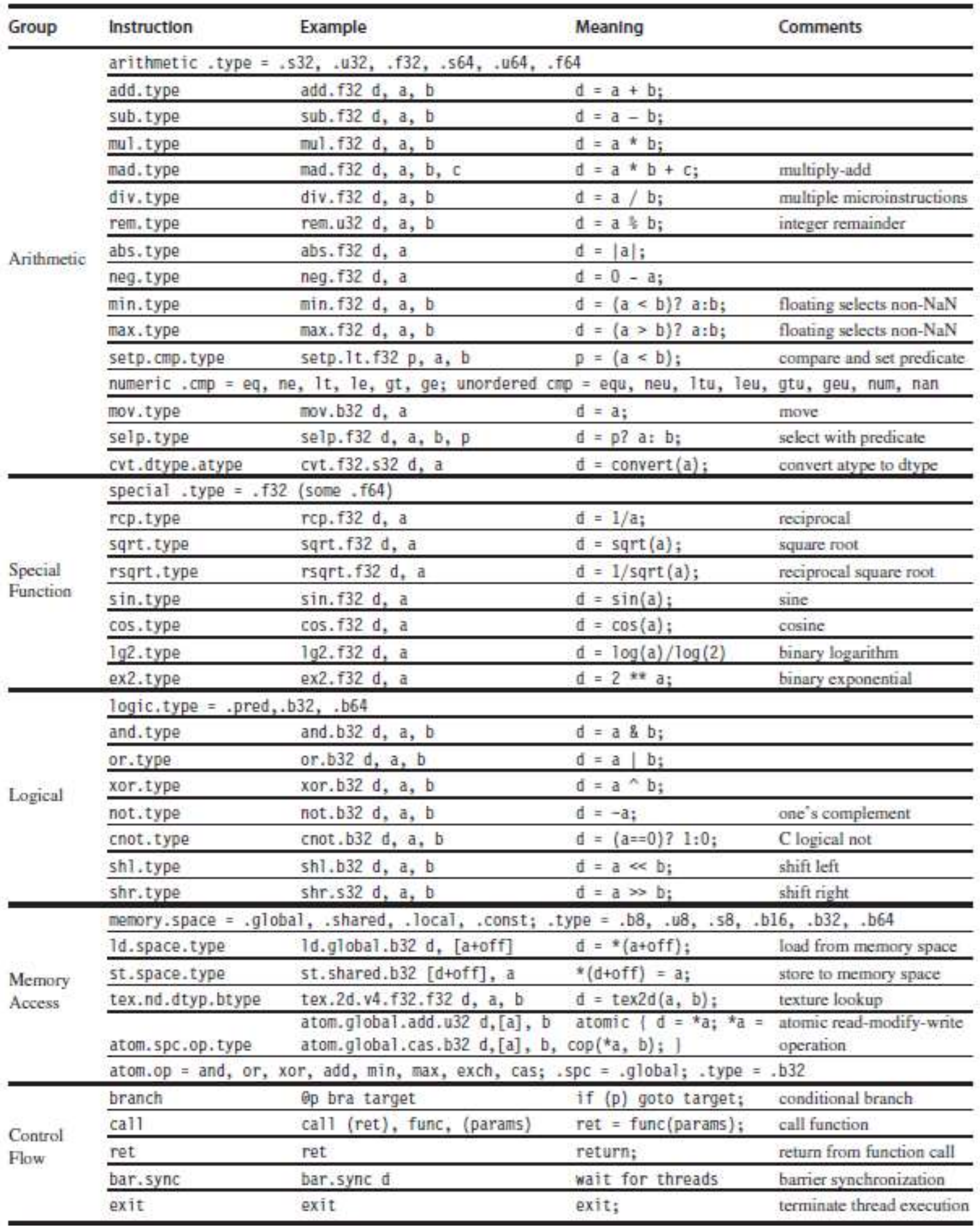
- Unlike most system processors, NVIDIA compilers target PTX, an abstraction of the hardware instruction set that provides:
  - A stable instruction set for compilers
  - Compatibility across generations of GPUs
  - Virtual registers (physical register allocation happens at load time)
- PTX instructions describe operations on a single CUDA thread and usually map one-to-one with hardware instructions, but one PTX instruction can expand to many machine instructions, and vice versa.

PTX Instruction Format


```
opcode.type d, a, b, c;
```

Type	.type Specifier
Untyped bits 8, 16, 32, and 64 bits	`.b8, .b16, .b32, .b64`
Unsigned integer 8, 16, 32, and 64 bits	`.u8, .u16, .u32, .u64`
Signed integer 8, 16, 32, and 64 bits	`.s8, .s16, .s32, .s64`
Floating Point 16, 32, and 64 bits	`.f16, .f32, .f64`

Basic PTX Instructions



Group	Instruction	Example	Meaning	Comments
	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a – b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.type	rem.u32 d, a, b	d = a % b;	integer remainder
Arithmetic	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 - a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
Special Function	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential
	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
Logical	xor.type	xor.b32 d, a, b	d = a ^ b;	
	not.type	not.b32 d, a, b	d = ~a;	one's complement
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.type	shl.b32 d, a, b	d = a << b;	shift left
	shr.type	shr.s32 d, a, b	d = a >> b;	shift right
	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
Memory Access	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
Control Flow	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

Example

```
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

<One iteration>

```
shl.u32 R8, blockIdx, 9      ; Thread Block ID * Block size (512 or 2^9)
add.u32 R8, R8, threadIdx    ; R8 = i = my CUDA Thread ID
shl.u32 R8, R8, 3            ; byte offset
ld.global.f64 RD0, [X+R8]    ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]    ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4        ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, RD0, RD2        ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0    ; Y[i] = sum (X[i]*a + Y[i])
```

Address Coalescing

- To regain the efficiency of sequential (unit-stride) data transfers, GPUs include special Address Coalescing hardware that:
  - Recognizes when SIMD Lanes within a thread are collectively issuing sequential addresses
  - Notifies the Memory Interface Unit to request a block transfer of 32 sequential words
- For optimal performance, GPU programmers must ensure that adjacent CUDA Threads access nearby addresses at the same time that can be coalesced into one or a few memory or cache blocks.

Conditional Branching in GPUs

Branch Handling Mechanisms

- Explicit predicate registers
- Internal masks
- Branch synchronization stack
- Instruction markers

These mechanisms manage when a branch diverges into multiple execution paths and when paths converge.

Branch Efficiency

For equal length paths:

- IF-THEN-ELSE: 50% efficiency
- Doubly nested IF: 25% efficiency
- Triply nested IF: 12.5% efficiency

When all lanes agree on the branch condition, the processor can skip over unused code blocks entirely.

Conditional Branch Example

Original Code

```
if (X[i] != 0)
  X[i] = X[i] - Y[i];
else  X[i] = Z[i];
```

PTX Code

```
ld.global.f64 RD0, [X+R8]    ; RD0 = X[i]
setp.neq.s32 P1, RD0, #0     ; P1 is predicate register 1
@!P1, bra ELSE1, *Push       ; Push old mask, set new mask bits
                             ; if P1 false, go to ELSE1
ld.global.f64 RD2, [Y+R8]    ; RD2 = Y[i]
sub.f64 RD0, RD0, RD2        ; Difference in RD0
st.global.f64 [X+R8], RD0    ; X[i] = RD0
@P1, bra ENDIF1, *Comp       ; complement mask bits
                             ; if P1 true, go to ENDIF1
ELSE1:  ld.global.f64 RD0, [Z+R8]    ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0    ; X[i] = RD0
ENDIF1: <next instruction>, *Pop     ; pop to restore old mask
```

*Push, *Comp, and *Pop indicate branch synchronization markers inserted by the PTX assembler to manage mask registers.