Where Is That Instruction?
How To Implement "Missing" Vertex Shader Instructions

Contact: Matthias Wloka (mwloka@nvidia.com)

1. Introduction 1
2. Constant ONE Generation 2
3. Raise to the Power 2
4. Compute the Fractional Part 2
5. Absolute Value 3
6. Division 3
7. Square Root 3
8. Set on Less Or Equal 3
9. Set on Greater 3
10. Set on Equal 3
11. Set on Not Equal 4
12. Clamp to [0, 1] Range 4
13. Compute the Floor 4
14. Compute the Ceiling 4
15. Vector Cross Product 4
16. Multiply Vector with Transpose Matrix 4
17. High Precision Cosine and Sine 5
18. Table Look-Up Cosine and Sine 5
19. Taylor-Series Expansion Cosine and Sine 6
20. Another Simultaneous Cosine and Sine 6
21. High-Precision EXP2 7
22. High-Precision LOG2 7
23. If-Then-Else 8
24. Signum Function 8
25. Smallest Magnitude Vector 8
26. Put Your Favorite Operation Here And Tell Us About It 9

1. Introduction

Vertex shaders vs1.1 only know 17 different instructions. Because this instruction set is so small, you may find that some operations that you consider basic are missing. For example, there is no sine instruction. This document shows how to implement these "missing" operations. More often than not, cunning use of the existing instructions, source-negation, and swizzling implement "missing" operations using surprisingly few instructions. This document collects these useful code-fragments in one place.

If you know how to efficiently compute your favorite operation, or know a faster or better way to compute one of the operations described here, please send a note to the contact above. Several of the code-snippets listed here were contributed by independent developers.

2. Constant ONE Generation

Why generate a constant one on the fly, you may ask. Because vertex shaders cannot read from two different constant addresses in the same instruction: For example,

```
DEF c1, 1.0f, 0.0f, 0.0f, 0.0f
MAD r0, r1, c0, c1.x
```

is illegal. So, if trying to add one to c0*r1, you need to waste an instruction slot to load a constant one into a register. Here is the obvious way of doing it.

```
DEF c1, 1.0f, 0.0f, 0.0f, 0.0f
MOV r0.x, c1.x
MAD r0, r1, c0, r0.x
```

Using

```
SGE r0, r0, r0
MAD r0, r1, c0, r0
```

instead has the following advantages: you save space in the constant memory that you can fill with something more useful, and more important you enable full optimization as the compiler/driver can figure out that you are generating a constant one, and not loading an arbitrary constant!

Similarly,

```
SLT r0, r0, r0
```

Generates constant zero in any of the components of r0 (use write masks).

3. Raise to the Power

This is actually in the instruction set, except it is called **`LIT`** and has a few restrictions: the power is clamped to [–128, +128], the base must be positive, and the accuracy is only good for 8-bit color output.

```
; compute scalar r0.z = r1.x^r1.y

LIT r0.z, r1.xxyy       ; r1.x must be greater than zero
```

4. Compute the Fractional Part

This instruction is provided as a macro in DX8. We do not recommend the use of macros: they obscure optimization opportunities.

```
; compute the fractional part of r1.xyzw and store in r0.xyzw

EXPP r0.y, r1.x
MOV  r0.x, r0.y
EXPP r0.y, r1.z
MOV  r0.z, r0.y
EXPP r0.y, r1.w
MOV  r0.w, r0.y
EXPP r0.y, r1.y
```

Only use the parts that you need!

5. Absolute Value

```
; r0 = |r1|

MAX r0, r1, -r1
```

6. Division

```
; scalar r0.x = r1.x/r2.x

RCP r0.x, r2.x
MUL r0.x, r1.x, r0.x
```

7. Square Root

```
; scalar r0.x = sqrt(r1.x)

RSQ r0.x, r1.x          ; using x/sqrt(x) = sqrt(x) is higher
MUL r0.x, r0.x, r1.x    ; precision than 1/( 1/sqrt(x) )
```

8. Set on Less or Equal

```
; r0 = (r1 <= r2) ? 1 : 0

SGE r0, -r1, -r2
```

9. Set on Greater

```
; r0 = (r1 r2) ? 1 : 0

SLT r0, -r1, -r2
```

10. Set on Equal

```
; r0 = (r1 == r2) ? 1 : 0
```

```
SGE r0, -r1, -r2
SGE r2,  r1,  r2
MUL r0,  r0,  r2
```

11. Set on Not Equal

```
; r0 = (r1 != r2) ? 1 : 0

SLT r0,  r1,  r2
SLT r2, -r1, -r2
ADD r0,  r0,  r2
```

12. Clamp to [0, 1] Range

```
; compute r0 = (r0 < 0) ? 0 : (r0 1) ? 1 : r0

DEF c0, 0.0f, 1.0f, 0.0f, 0.0f
MAX r0, r0, c0.x
MIN r0, r0, c0.y
```

13. Compute the Floor

```
; scalar r0.y = floor(r1.y)

EXPP r0.y, r1.y
ADD  r0.y, r1.y, - r0.y
```

14. Compute the Ceiling

```
; scalar r0.y = ceiling(r1.y)

EXPP r0.y, -r1.y
ADD  r0.y, r1.y, r0.y
```

15. Vector Cross Product

```
; r0 = r1 x r2 (3-vector cross-product)

MUL r0,  r1.yzxw, r2.zxyw
MAD r0, -r2.yzxw, r1.zxyw, r0
```

16. Multiply Vector with Transpose Matrix

```
; c0–c3 is matrix to transpose and multiply r1 with

MUL r0, c0, r1.x
MAD r0, c1, r1.y, r0
MAD r0, c2, r1.z, r0
MAD r0, c3, r1.w, r0
```

17. High Precision Cosine and Sine

```
; scalar r0.x = cos(r1.x)

DEF c0,  0.0f,   0.5f, 1.0f,  0.0f
DEF c1,  0.25f, -9.0f, 0.75f, 1.0f/(2.0f*PI)

DEF c2, 24.9808039603f, -24.9808039603f, -60.1458091736f, 60.1458091736
DEF c3, 85.4537887573f, -85.4537887573f, -64.9393539429f, 64.9393539429
DEF c4, 19.7392082214f, -19.7392082214f, - 1.0f,           1.0f

MUL  r1.x,    c1.w,    r1.x         ; normalize input
EXPP r1.y,    r1.x                  ; and extract fraction
SLT  r2.x,    r1.y,    c1           ; range check: 0.0 to 0.25
SGE  r2.yz,   r1.y,    c1           ; range check: 0.75 to 1.0
DP3  r2.y,    r2,      c4.zwzw      ; range check: 0.25 to 0.75
ADD  r0.xyz, -r1.y,    c0           ; range centering
MUL  r0,      r0,      r0
MAD  r1,      c2.xyxy, r0, c2.zwzw  ; start power series
MAD  r1,      r1,      r0, c3.xyxy
MAD  r1,      r1,      r0, c3.zwzw
MAD  r1,      r1,      r0, c4.xyxy
MAD  r1,      r1,      r0, c4.zwzw
DP3  r0.x,    r1,     -r2           ; range extract
```

And similarly:

```
; scalar r0.x = sin(r1.x)

DEF c0,  0.0f,   0.5f, 1.0f,  0.0f
DEF c1,  0.25f, -9.0f, 0.75f, 1.0f/(2.0f*PI)

DEF c2, 24.9808039603f, -24.9808039603f, -60.1458091736f, 60.1458091736
DEF c3, 85.4537887573f, -85.4537887573f, -64.9393539429f, 64.9393539429
DEF c4, 19.7392082214f, -19.7392082214f, - 1.0f,           1.0f

MAD  r1.x,   c1.w,    r1.x, -c1.x   ; only difference from COS!
EXPP r1.y,    r1.x                  ; and extract fraction
SLT  r2.x,    r1.y,    c1           ; range check: 0.0 to 0.25
SGE  r2.yz,   r1.y,    c1           ; range check: 0.75 to 1.0
DP3  r2.y,    r2,      c4.zwzw      ; range check: 0.25 to 0.75
ADD  r0.xyz, -r1.y,    c0           ; range centering
MUL  r0,      r0,      r0
MAD  r1,      c2.xyxy, r0, c2.zwzw  ; start power series
MAD  r1,      r1,      r0, c3.xyxy
MAD  r1,      r1,      r0, c3.zwzw
MAD  r1,      r1,      r0, c4.xyxy
MAD  r1,      r1,      r0, c4.zwzw
DP3  r0.x,    r1,     -r2           ; range extract
```

18. Table Look-Up Cosine and Sine

Using a table look-up into constant memory and linearly interpolating between the look-ups is one of the fastest ways to compute the cosine and sine values. Using constant

look-up (i.e., not using linear interpolation) saves another 3 instructions. The number of table-entries can be varied depending on required accuracy and available constant memory space. The example listed here uses a table-size of 5 for clarity; expect to use tables of size 16 to 32 for reasonable accuracy.

```
; scalar r0.x = cos(r1.x), r0.y = sin(r1.x)

DEF c0,  1.f, 5.f, 0.f, 1.f/(2.f*PI)

DEF c1,  SIN(0.0f * 2.f*PI), COS(0.0f * 2.f*PI), 0.f, 0.f
DEF c2,  SIN(0.2f * 2.f*PI), COS(0.2f * 2.f*PI), 0.f, 0.f
DEF c3,  SIN(0.4f * 2.f*PI), COS(0.4f * 2.f*PI), 0.f, 0.f
DEF c4,  SIN(0.6f * 2.f*PI), COS(0.6f * 2.f*PI), 0.f, 0.f
DEF c5,  SIN(0.8f * 2.f*PI), COS(0.8f * 2.f*PI), 0.f, 0.f
DEF c6,  SIN(1.0f * 2.f*PI), COS(1.0f * 2.f*PI), 0.f, 0.f

MUL  r0.x,  c0.w, r1.x        ; normalize input
EXPP r1.y,  r0.x              ; to [0..1] range
MAD  r0.w,  c0.y,  r1.y, c0.x ; scale to table-size and add table-base
MOV  a0.x,  r0.w
EXPP r1.yw, r0.w              ; use fractional part for lerp
ADD  r1.x,  r1.w, -r1.y       ; use (1 – fractional part) for lerp

; Lerp
MUL r0.xy, c[a0.x    ].xy, r1.x
MAD r0.xy, c[a0.x + 1].xy, r1.y, r0.xy
```

19. Taylor-Series Expansion Cosine and Sine

If you are less interested in precision, but need more speed and do not have space in the constant memory, this simple Taylor-series expansion might do the trick. It computes cos() and sin() at the same time.

```
; scalar r0.x = cos(r1.x), r0.y = sin(r1.x)

DEF c0, PI,    1.f/2.f, 2.f*PI,     1.f/(2.f*PI)
DEF c1, 1.0f, -1.f/2.f, 1.f/24.f,  -1.f/720.f
DEF c2, 1.0f, -1.f/6.f, 1.f/120.f, -1.f/5040.f

MAD  r0.x,  r1.x, c0.w,  c0.y ; bring argument into –pi, .., +pi range
EXPP r0.y,  r0.x
MAD  r0.x,  r0.y, c0.z, -c0.x
DST  r2.xy, r0.x, r0.x        ; generate 1, (r0.x)^2, .. (r0.x)^6
MUL  r2.z,  r2.y, r2.y
MUL  r2.w,  r2.y, r2.z
MUL  r0,    r2,   r0.x        ; generate r0.x, (r0.x)^3, .., (r0.x)^7
DP4  r0.y,  r0,   c2          ; compute sin(r0.x)
DP4  r0.x,  r2,   c1          ; compute cos(r0.x)
```

20. Another Simultaneous Cosine and Sine

This vertex shader promises to be higher precision than the Taylor-series cosine/sine, but faster than the separate high-precision Cosine/Sine computation.

```
; scalar r0.x = cos(r1.x)
; scalar r0.y = sin(r1.x)

DEF c0,  0.25f,          0.5f,            0.75f,          1.0f
DEF c1,-24.9808039603f, 60.1458091736f, -85.4537887573f, 64.9393539429f
DEF c2,-19.7392082214f,  1.0f,          - 1.0f,           1.0f/(2*PI)

MUL  r1.x,   c2.w,     r1.x     ; Normalize input,
EXPP r1.y,   r1.x               ; and extract fraction.
SLT  r2,     r1.yyyy,  c0       ; Range check, one each to indicate:
ADD  r2.yzw, r2.xyzw, -r2.xxyz  ; [0,.25),[.25,.5),[.5,.75),[.75,1.0)
DP3  r1.z,   r2.yzwx,  c0.yywx  ; Calc shift for cos.
DP4  r1.w,   r2,       c0.xxzz  ; Calc shift for sin.
ADD  r0.xz,  r1.yyyy, -r1.zzww  ; x for cos, z for sin
MUL  r0.xz,  r0.xxzz,  r0.xxzz           ; [ x^2,   #, z^2,   #]
MUL  r0.yw,  r0.xxzz,  r0.xxzz           ; [ x^2, x^4, z^2, z^4]
MAD  r1,     c1.xyxy,  r0.yyww, c1.zwzw
MAD  r1,     r1,       r0.yyww, c2.xyxy
MAD  r1.xz,  r1,       r0.xxzz, r1.yyww
DP4  r0.x,   r2,       c2.yzzy           ; Sign for cos.
DP4  r0.y,   r2,       c2.yyzz           ; Sign for sin.
MUL  r0.xy,  r0.xyww,  r1.xzww
```

21. High-Precision EXP2

Warning! Do not confuse the DX8 instruction **`EXPP`** with the DX8 macro **`EXP`**: The **`EXPP`** instruction computes a 10-bit precision exp2() function in a single tick; the macro **`EXP`** computes a full-precision exp2() function in many more ticks (!).

```
; compute scalar r0.z = exp2(r1.z)

DEF c0, 1.00000000,    -6.93147182e-1, 2.40226462e-1, -5.55036440e-2
DEF c1, 9.61597636e-3, -1.32823968e-3, 1.47491097e-4, -1.08635004e-5

EXPP r0.xy, r1.z
DST  r1,    r0.y,    r0.y
MUL  r1,    r1.xzyw, r1.xxxy        ; 1,x,x^2,x^3
DP4  r0.z,  r1,      c0
DP4  r0.w,  r1,      c1
MUL  r1.y,  r1.z,    r1.z           ; compute x^4
MAD  r0.w,  r0.w,    r1.y, r0.z     ; add the first to the last 4 terms
RCP  r0.w,  r0.w
MUL  r0.z,  r0.w,    r0.x           ; multiply by 2^I
```

22. High-Precision LOG2

Warning! Do not confuse the DX8 instruction **`LOGP`** with the DX8 macro **`LOG`**: The instruction **`LOGP`** computes a 10-bit precision log2() function in a single tick; the macro **`LOG`** computes a full-precision log2() function in 12 ticks (!).

```
; scalar r0.x = log2(r1.x)
```

```
DEF c0, 1.44268966,    -7.21165776e-1, 4.78684813e-1, -3.47305417e-1
DEF c1, 2.41873696e-1, -1.37531206e-1, 5.20646796e-2, -9.31049418e-3
DEF c2, 1.0f,           0.0f,          0.0f,           0.0f

LOGP r0.x, r1.x
ADD  r0.y, r0.x, -c2.x        ; subtract 1.0
DST  r1, r0.y,    r0.y
MUL  r1, r1.xzyw, r1.xxxy     ; 1,x,x^2,x^3
DP4  r0.z, r1,    c0
DP4  r0.w, r1,    c1
MUL  r1.y, r1.z,  r1.z        ; compute x^4
MAD  r0.w, r0.w,  r1.y, r0.z  ; and add the first 4 to the last 4 terms
MAD  r0.x, r0.w,  r0.y, r0.x  ; exponent add
```

23. If-Then-Else

```
; compute r0 = (r1 >= r2) ? r3 : r4

SGE r0, r1,  r2         ; one if (r1 >= r2) holds, zero otherwise
ADD r1, r3, -r4
MAD r0, r0, r1, r4      ; r0 = r0*(r3-r4) + r4 = r0*r3 + (1-r0)*r4
```

The above solution relies on (r3-r4) being a sensible result; if (r3-r4) is impossible to compute (for example, because r3 is infinity), then the following solution is preferable:

```
DEF c0, 1.0f, 0.0f, 0.0f, 0.0f

SGE r0, r1,  r2         ; one if (r1 >= r2) holds, zero otherwise
ADD r1, c0, -r0
MUL r1, r1,  r4
MAD r0, r0,  r3, r1     ; r0 = r0*r3 + (1-r0)*r4
```

24. Signum Function

```
;               1 if (r0  0)
; compute r0 =  0 if (r0 == 0)
;              -1 if (r0  < 0)

DEF c0, 0.0f, 0.0f, 0.0f, 0.0f

SLT r1,  r0,  c0
SLT r0, -r0,  c0
ADD r0,  r0, -r1
```

25. Smallest Magnitude Vector

```
; compute   r1.x = 1 if (|r0.x| <= |r0.y| && |r0.x| <  |r0.z|)
;                  0 otherwise
;           r1.y = 1 if (|r0.y| <  |r0.x| && |r0.y| <= |r0.z|)
;                  0 otherwise
;           r1.z = 1 if (|r0.z| <= |r0.x| && |r0.z| <  |r0.y|)
;                  0 otherwise
```

```
MAX r1, r0, -r0
SLT r2.xyz, r1.xyzw, r1.zxyw
SGE r1.xyz, r1.yzxw, r1.xyzw
MUL r1.xyz, r2, r1
```

26. Put Your Favorite Operation Here and Tell Us about It