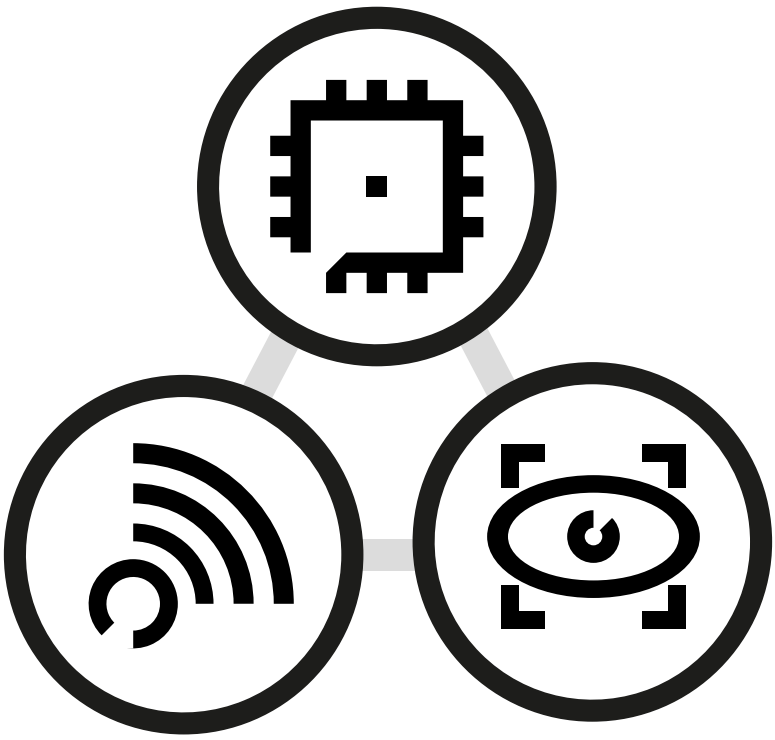
PowerVR Low Level GLSL Optimisation

Revision: 1.0
12/05/2021
Public



Published: 12/05/2021-19:15

Contents

1. General Overview

Introduction to Low Level GLSL Optimisation

This guide describes ways to optimise GLSL code for PowerVR Rogue architecture. These optimisations are low-level, and therefore can be used to get the last ten percent of performance boost from the hardware. Before using these techniques, it is essential to make sure that the most optimal algorithms are being used, and that the GPU is well utilised.

The effectiveness of these optimisations may vary depending on the exact compiler architecture used. It is always worth checking if optimisations result in a performance improvement on the target platform. Throughout the document there are examples of USC instructions the GLSL code compiles to.

This document is based on the following:

Low-Level Thinking in High-Level Shading Languages, by Emil Persson.

Low-Level Shader Optimisation for Next-Gen and DX11, by Emil Persson.

PowerVR Rogue USC

Shader performance on PowerVR Rogue architecture GPUs depends on the number of cycles it takes to execute a shader. Depending on the configuration, the PowerVR Rogue architecture delivers a variety of options for executing multiple instructions in the USC ALU pipeline within a single cycle. Some of these instructions are explained in more detail further along in this guide.

For example, it is possible to execute all of the following in one cycle:

- Two F16 `SOP` instructions
- The F32 <-> F16 conversions
- The `MOV/OUTPUT/PACK` instruction.

As can all of these in one cycle:

- An FP32 `MAD`
- An FP32/INT32 `MAD/UNPACK` instruction
- A test (conditional) instruction
- The `MOV/OUTPUT/PACK` instruction.

If there is bitwise work to be done, these can all be executed in one cycle too:

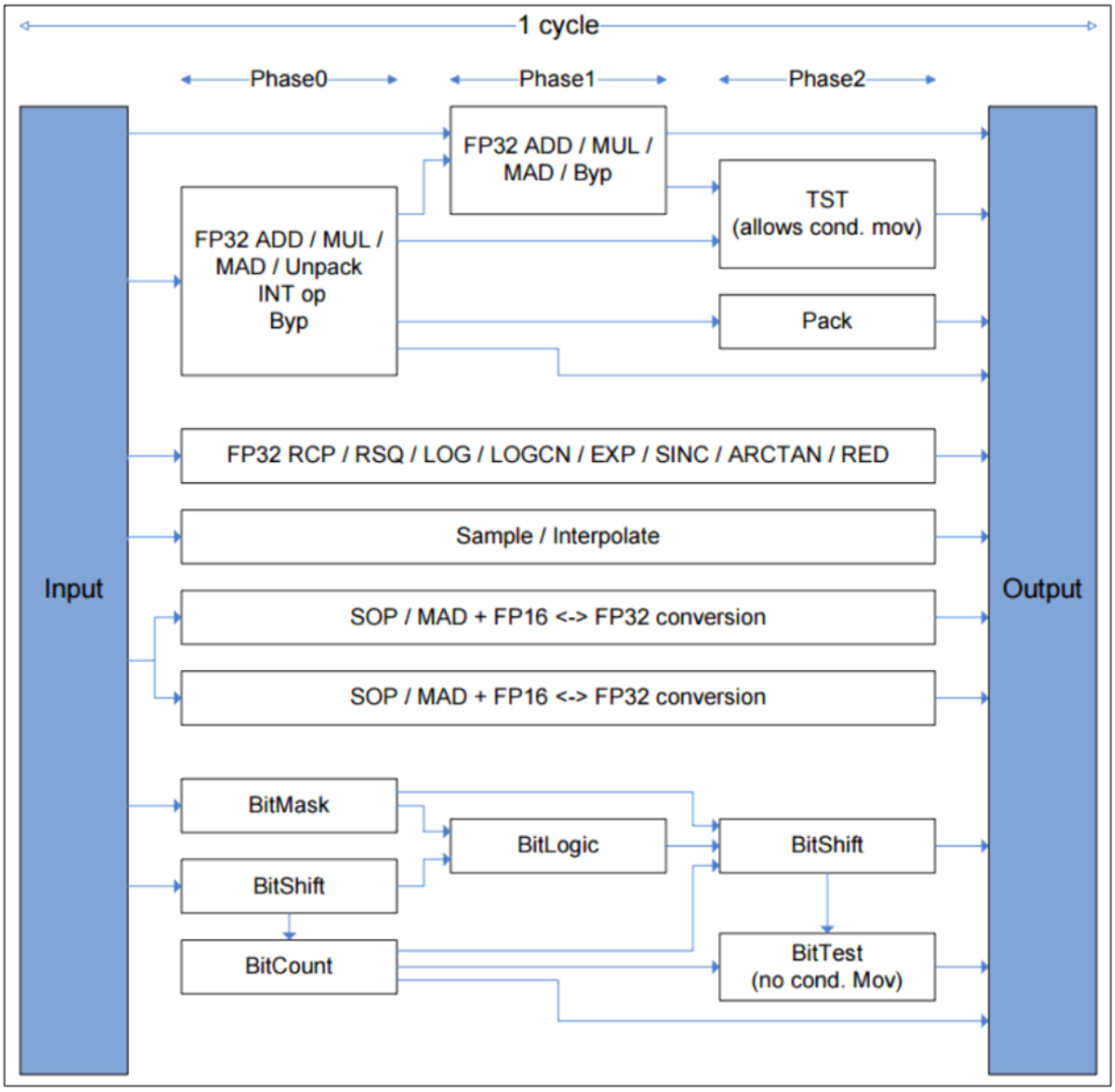
- A bitwise SHIFT/COUNT
- A bitwise logical operation
- A bitwise SHIFT

- A test (conditional) instruction
- The `MOV/OUTPUT/PACK` instructions.

Other possibilities for execution in one cycle include:

- A single complex operation (such as `RCP`) and a `MOV/OUTPUT/PACK` instruction
- An interpolate/sample instruction plus the usual `MOV/OUTPUT/PACK` instruction.

As shown in the diagram below, it is best to use all stages in one route in the pipeline below to make full use of the ALU. Therefore, GLSL instructions should be arranged in that way.



Writing Expressions in MAD Form

To take advantage of the USC cores, it is essential to always write math expressions in Multiply-Add form (MAD form).

For example, changing the following expression to use the MAD form results in a 50% cycle cost reduction:

```
fragColor.x = (t.x + t.y) * (t.x - t.y); // two cycles
{sop, sop, sopmov}
{sop, sop}
-->
fragColor.x = t.x * t.x + (-t.y * t.y); // one cycle
{sop, sop}
```

Related Concepts

FP16 SOP/MAD Operation

Exploiting the SOP/MAD FP16 Pipeline

Division

It is usually better to write division math in reciprocal form as the reciprocal form is directly supported by an instruction (`RCP`).

Finishing math expressions' simplification can give further performance gains.

```
fragColor.x = (t.x * t.y + t.z) / t.x; // three cycles
{sop, sop, sopmov}
{frcp}
{sop, sop}
-->
fragColor.x = t.y + t.z * (1.0 / t.x); // two cycles
{frcp}
{sop, sop}
```

Sign

Originally the result of `sign(x)` would be:

```
-1 if x < 0,
1 if x > 0
```

and

```
0 if x == 0
```

However, if the last case is not needed it is better to use conditional form instead of `sign()`.

```
fragColor.x = sign(t.x) * t.y; // three cycles
{mov, pck, tstgez, mov}
{mov, pck, tstgez, mov}
{sop, sop}
-->
fragColor.x = (t.x >= 0.0 ? 1.0 : -1.0) * t.y; // two cycles
{mov, pck, tstgez, mov}
{sop, sop}
```

Reciprocal, RSqrt, and Sqrt

On the PowerVR Rogue architecture, the reciprocal operation is directly supported by an instruction (`RCP`):

```
fragColor.x = 1.0 / t.x; // one cycle
{frcp}
```

The same is true with the `inversesqrt()` function (`RSQ`):

```
fragColor.x = inversesqrt(t.x); // one cycle
{frsq}
```

However, `sqrt()` is implemented as: `1 / (1/sqrt(x))`.

This results in a two cycle cost:

```
fragColor.x = sqrt(t.x); // two cycles
{frsq}
{frcp}
```

A commonly used alternative: `x * 1/sqrt(x)` gives the same two cycle results.

```
fragColor.x = t.x * inversesqrt(t.x); // two cycles
{frsq}
{sop, sop}
```

The only case when it is better to use the above alternative is if the result is tested. In this case, the test instructions can fit into the second instruction.

```
fragColor.x = sqrt(t.x) > 0.5 ? 0.5 : 1.0; // three cycles
{frsq}
{frcp}
{mov, mov, pck, tstg, mov}
-->
fragColor.x = (t.x * inversesqrt(t.x)) > 0.5 ? 0.5 : 1.0; // two cycles
{frsq}
{fmul, pck, tstg, mov}
```

Abs, Neg, and Saturate

On PowerVR architecture, it is essential to use modifiers such as `abs()`, `neg()`, and `clamp(…, 0.0, 1.0)` (also known as `saturate()`) - being free in certain cases.

`abs()` and `neg()` are free if they are used on an input to an operation, in which case they are turned into a free modifier by the compiler. However, `saturate()` turns into a free modifier when used on the output of an operation.

Note: Complex and sampling/interpolation instructions are exceptions to this rule. `saturate()` is not free when used on a texture sampling output, or on a complex instruction output. When these functions are not used accordingly, they may introduce additional `MOV` instructions which may increase the cycle count of the shaders.

It is also beneficial to use `clamp(…, 0.0, 1.0)` instead of `min(…, 1.0)` and `max(…, 0.0)`. This changes a test instruction into a saturate modifier:

```
fragColor.x = abs(t.x * t.y); // two cycles
{sop, sop}
{mov, mov, mov}
-->
fragColor.x = abs(t.x) * abs(t.y); // one cycle
{sop, sop}
fragColor.x = -dot(t.xyz, t.yzx); // three cycles
{sop, sop, sopmov}
{sop, sop}
{mov, mov, mov}
-->
fragColor.x = dot(-t.xyz, t.yzx); // two cycles
```

```
{sop, sop, sopmov}
{sop, sop}
fragColor.x = 1.0 - clamp(t.x, 0.0, 1.0); // two cycles
{sop, sop, sopmov}
{sop, sop}
-->
fragColor.x = clamp(1.0 - t.x, 0.0, 1.0); // one cycle
{sop, sop}
fragColor.x = min(dot(t, t), 1.0) > 0.5 ? t.x : t.y; // five cycles
{sop, sop, sopmov}
{sop, sop}
{mov, fmad, tstg, mov}
{mov, mov, pck, tstg, mov}
{mov, mov, tstz, mov}
-->
fragColor.x = clamp(dot(t, t), 0.0, 1.0) > 0.5 ? t.x : t.y; // four cycles
{sop, sop, sopmov}
{sop, sop}
{fmad, mov, pck, tstg, mov}
{mov, mov, tstz, mov}
```

However, it is sensible to be wary of complex functions, as they are translated into multiple operations. Therefore in this case it matters where the modifiers are placed.

For example, `normalize()` is broken down into:

```
vec3 normalize( vec3 v )
{
    return v * inverssqrt( dot( v, v ) );
}
```

In this case it is best to negate one of the inputs of the final multiplication rather than the inputs in all cases, or create a temporary negated input:

```
fragColor.xyz = -normalize(t.xyz); // six cycles
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, mov}
-->
fragColor.xyz = normalize(-t.xyz); // seven cycles
{mov, mov, mov}
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, mov}
```

2. Transcendental functions

Exp and Log

On PowerVR Rogue architecture, the `2^n` operation is directly supported by an instruction (`EXP`):

```
fragColor.x = exp2(t.x); // one cycle
{fexp}
```

The same is true with the `log2()` function (`LOG`):

```
fragColor.x = log2(t.x); // one cycle
{flog}
```

`exp` is implemented as:

```
float exp2( float x )
{
    return exp2(x * 1.442695); // two cycles
    {sop, sop}
    {fexp}
}
```

`log` is implemented as:

```
float log2( float x )
{
    return log2(x * 0.693147); // two cycles
    {sop, sop}
    {flog}
}
```

`pow(x, y)` is implemented as:

```
float pow( float x, float y )
{
    return exp2(log2(x) * y); // three cycles
    {flog}
    {sop, sop}
    {fexp}
}
```

Sin, Cos, Sinh, and Cosh

Sin, cos, sinh, and cosh on PowerVR architecture have a reasonably low cost of four cycles.

This is broken down as:

- Two cycles of reduction
- `sinc`
- One conditional.

```
fragColor.x = sin(t.x); // four cycles
{fred}
{fred}
{fsinc}
```

```
{fmul, mov} // plus conditional

fragColor.x = cos(t.x); // four cycles
{fred}
{fred}
{fsinc}
{fmul, mov} // plus conditional

fragColor.x = cosh(t.x); // three cycles
{fmul, fmul, mov, mov}
{fexp}
{sop, sop}

fragColor.x = sinh(t.x); // three cycles
{fmul, fmul, mov, mov}
{fexp}
{sop, sop}
```

Asin, Acos, Atan, Degrees, and Radians

If the math expressions' simplifications are completed, then these functions are usually not needed. Therefore they do not map to the hardware exactly. This means that these functions have a very high cost, and should be avoided at all times.

`asin()` costs a massive 67 cycles:

```
fragColor.x = asin(t.x); // 67 cycles
// USC code omitted due to length
```

`acos()` costs a massive 79 cycles:

```
fragColor.x = acos(t.x); // 79 cycles
// USC code omitted due to length
```

`atan()` is still costly, but it could be used if needed:

```
fragColor.x = atan(t.x); // 12 cycles (lots of conditionals)
// USC code omitted due to length
```

While degrees and radians take only one cycle, they can usually be avoided if only radians are used:

```
fragColor.x = degrees(t.x); // one cycle
{sop, sop}
fragColor.x = radians(t.x); // one cycle
{sop, sop}
```

3. Intrinsic functions

Vector*Matrix

The vector * matrix multiplication operation has quite a reasonable cost, despite the number of calculations that need to happen.

However, optimisations such as taking advantage of knowing that `w` is 1, do not reduce the cost.

```
fragColor = t * m1; // 4x4 matrix, eight cycles
{mov}
{wdf}
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
fragColor.xyz = t.xyz * m2; // 3x3 matrix, four cycles
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop}
```

Mixed Scalar/Vector Math

`normalise()`/`length()`/`distance()`/`reflect()` functions usually contain a lot of function calls inside them such as `dot()`. There is an advantage to knowing how these functions are implemented.

For example, if it is known that two operations have a shared sub-expression, then it is possible to reduce the cycle count. However, this only happens if the input order allows it.

```
fragColor.x = length(t-v); // seven cycles
fragColor.y = distance(v, t);
{sopmad, sopmad, sopmad, sopmad}
{sop, sop, sopmov}
{sopmad, sopmad, sopmad, sopmad}
{sop, sop, sopmov}
{sop, sop}
{frsq}
{frcp}
-->
fragColor.x = length(t-v); // nine cycles
fragColor.y = distance(t, v);
{mov}
{wdf}
{sopmad, sopmad, sopmad, sopmad}
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{frsq}
{frcp}
{mov}
```

Manually expanding these complex instructions can sometimes help the compiler optimise the code:

```
fragColor.xyz = normalize(t.xyz); // six cycles
```

```
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, mov}
-->
fragColor.xyz = inversesqrt( dot(t.xyz, t.xyz) ) * t.xyz; // five cycles
{sop, sop, sopmov}
{sop, sop}
{frsq}
{sop, sop}
{sop, sop}
```

Also, in expanded form it is possible to take advantage of grouping vector and scalar instructions together:

```
fragColor.xyz = 50.0 * normalize(t.xyz); // seven cycles
{fmul, mov}
{fmad, mov}
{fmad, mov}
{frsq}
{fmul, fmul, mov, mov}
{fmul, fmul, mov, mov}
{sop, sop}
-->
fragColor.xyz = (50.0 * inversesqrt( dot(t.xyz, t.xyz) )) * t.xyz; // six cycles
{sop, sop, sopmov}
{sop, sop}
{frsq}
{sop, sop, sopmov}
{sop, sop}
{sop, sop}
```

The following list shows what the complex instructions can be expanded to.

`cross()` can be expanded to:

```
vec3 cross( vec3 a, vec3 b )
{
    return vec3( a.y * b.z - b.y * a.z,
                 a.z * b.x - b.z * a.x,
                 a.x * b.y - b.y * a.y );
}
```

`distance()` can be expanded to:

```
float distance( vec3 a, vec3 b )
{
    vec3 tmp = a - b;
    return sqrt( dot(tmp, tmp) );
}
```

`dot()` can be expanded to:

```
float dot( vec3 a, vec3 b )
{
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

`faceforward()` can be expanded to:

```
vec3 faceforward( vec3 n, vec3 I, vec3 Nref )
{
    if( dot( Nref, I ) < 0 )
    {
      return n;
    }
    else
    {
      return -n:
    }
}
```

`length()` can be expanded to:

```
float length( vec3 v )
{
    return sqrt( dot(v, v) );
}
```

`normalize()` can be expanded to:

```
vec3 normalize( vec3 v )
{
    return v / sqrt( dot(v, v) );
}
```

`reflect()` can be expanded to:

```
vec3 reflect( vec3 N, vec3 I )
{
    return I - 2.0 * dot(N, I) * N;
}
```

`refract()` can be expanded to:

```
vec3 refract( vec3 n, vec3 I, float eta )
{
    float k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I));
    if (k < 0.0)
        return 0.0;
    else
        return eta * I - (eta * dot(N, I) + sqrt(k)) * N;
}
```

Operation Grouping

It is better to group scalar and vector operations separately. This way the compiler can pack more operations into a single cycle.

```
fragColor.xyz = t.xyz * t.x * t.y * t.wzx * t.z * t.w; // seven cycles
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop}
{sop, sop, sopmov}
{sop, sop}
-->
fragColor.xyz = (t.x * t.y * t.z * t.w) * (t.xyz * t.wzx); // four cycles
{sop, sop, sopmov}
{sop, sop, sopmov}
{sop, sop}
{sop, sop}
```

4. FP16 overview

FP16 Precision and Conversions

The FP16 pipeline works well when reduced precision is sufficient. However, it is advisable to always check whether the optimisations resulted in precision artefacts. When 16 bit float precision hardware is available, and shaders use mediump, then 16<->32-bit conversion is free using modifiers. This is because the USC ALU pipeline includes it.

When shaders do not use the 16-bit instructions or the hardware does not contain a 16-bit float pipeline, such as with early Rogue hardware, then the instructions just run on the regular 32-bit pipeline. As a result, no conversions happen.

FP16 SOP/MAD Operation

SOP stands for Sum of Products. The FP16 SOP/MAD pipeline is one of the strongest points of the PowerVR ALU pipeline. If used correctly, it makes it possible to pack more operations into a single cycle. This may result in increased performance and reduced power consumption.

The single cycle FP16 SOP/MAD operation can be described by the following pseudo code:

```
// Inputs:
a, b, c, d = any of {S0, S1, S2, S3, S4, S5}
z = min(s1, 1 - s0)
e, f, g, h = any of {S0, S1, S2, S3, S4, S5} or z

// Inputs only for the MAD pipeline:
v, w, x, y = any of {S0, S1, S2, S3, S4, S5}

// Operations to be performed
jop = any of {add, sub, min, max, rsub, mad}
kop = any of {add, sub, min, max, rsub}

// Either use the MAD or the SOP pipeline
if (jop == mad)
{
    // Two mad operations performed in parallel
    W0.e0 = a*e+v
    W1.e0 = b*f+x
    W0.e1 = c*g+w
    W1.e1 = d*h+y
}
else
{
    // Multiply the SOP inputs and perform the desired operation on the result
    // performed in parallel
    j = (a * e) jop (b * f)
    k = (c * g) kop (d * h)

    // Convert result to FP32 or keep the results as is
    if (rfmt(1) = 1)
    {
  w1 = toF32(k)
  w0 = toF32(j)
    }
    else if (rfmt(0) = 1) then
    {
  w0[31:16] = one of {j, a, b}
```

```
  w0[15:0]  = one of {k, c, d}
    }
    else
    {
  w0[31:16] = one of {k, c, d}
  w0[15:0]  = one of {j, a, b}
    }
}
```

It is also possible to apply various modifiers such as `abs()`, `negate()`, `clamp()`, or `oneminus()` to the inputs and `clamp()` to the outputs.

Exploiting the SOP/MAD FP16 Pipeline

The PowerVR Rogue architecture has a powerful FP16 pipeline optimised for common graphics operations. This section describes how to take advantage of this.

Important: Converting the inputs to FP16 and then converting the output to FP32 is free.

With `SOP`/`MAD` there are a number of options for execution in one cycle:

- Two `SOP`s
- Two `MAD`s
- One `MAD` and one `SOP`
- Four FP16 `MAD`s.

Here are some examples.

Executing four `MAD`s in one cycle:

```
fragColor.x = t.x * t.y + t.z;
fragColor.y = t.y * t.z + t.w;
fragColor.z = t.z * t.w + t.x;
fragColor.w = t.w * t.x + t.y;
{sopmad, sopmad, sopmad, sopmad}
```

`SOP` with a choice of an operation between the result of the multiplies:

```
fragColor.z = t.y * t.z OP t.w * t.x;
fragColor.w = t.x * t.y OP t.z * t.w;
```

where OP can be either an addition, a subtraction, a `min()` or a `max()` :

```
fragColor.z = t.y * t.z + t.w * t.x;
fragColor.z = t.y * t.z - t.w * t.x;
fragColor.z = min(t.y * t.z, t.w * t.x);
fragColor.z = max(t.y * t.z, t.w * t.x);
```

It is possible to apply either a negate, an `abs()`, or a `clamp()` (`saturate`) to all of the inputs:

```
fragColor.z = -t.y * abs(t.z) + clamp(t.w, 0.0, 1.0) * -t.x;
```

Finally, it is also possible to apply a `clamp()` (`saturate`) to the end result:

```
fragColor.z = clamp(t.y * t.z OP t.w * t.x, 0.0, 1.0);
fragColor.z = clamp(t.y * t.z + t.w, 0.0, 1.0);
```

After applying all this knowledge, the example below shows off the power of this pipeline by using everything in one cycle:

```
// one cycle
mediump vec4 fp16 = t;
highp vec4 res;
res.x = clamp(min(-fp16.y * abs(fp16.z), clamp(fp16.w, 0.0, 1.0) * abs(fp16.x)), 0.0, 1.0);
res.y = clamp(abs(fp16.w) * -fp16.z + clamp(fp16.x, 0.0, 1.0), 0.0, 1.0);
fragColor = res;
{sop, sop}
```