nVIDIA™

Programmable Texture Blending

John Spitzer
NVIDIA Corporation
jspitzer@nvidia.com

Where are we?

curved surfaces

vertex programs

per-vertex geometry & shading programs

setup rasterizer

You are here

texture shaders

tex fetch & filter

texture blending

per-pixel shading

frame-buffer

What is nvparse?

- A generalized compiler for NVIDIA extensions
- An easier way to use register combiners
- Does NOT replace the existing API, but is merely a layer on top of it
- Parser and the existing API can be used in concert (particularly when something like a constant color needs to be “tweaked”)

Standard OpenGL Texture Blending

Fragment Color

Texture Fetching

Tex0

Tex1

Tex2

Tex3

Texture Environment 0

Texture Environment 1

Texture Environment 2

Texture Environment 3

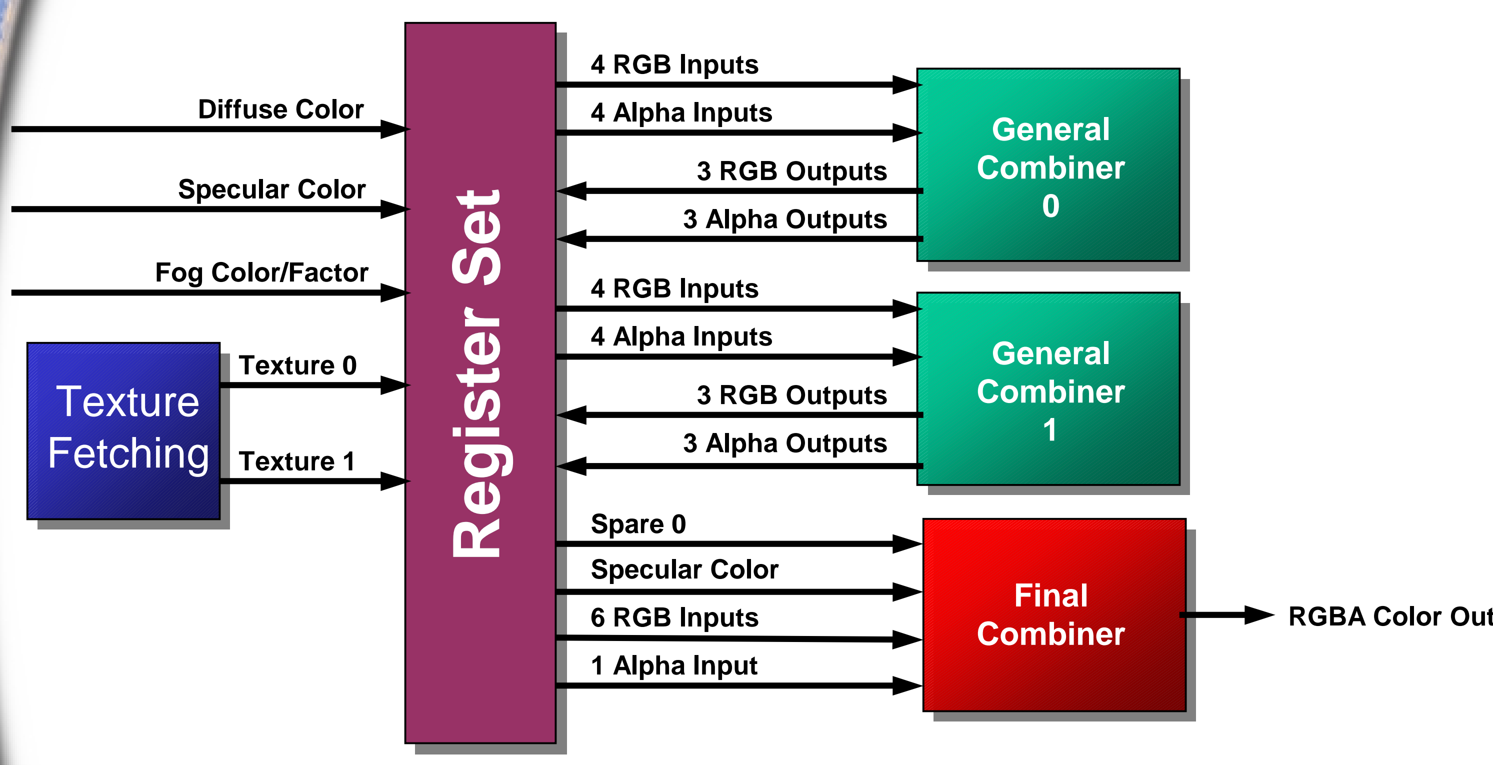
Specular Color

Specular Color Sum

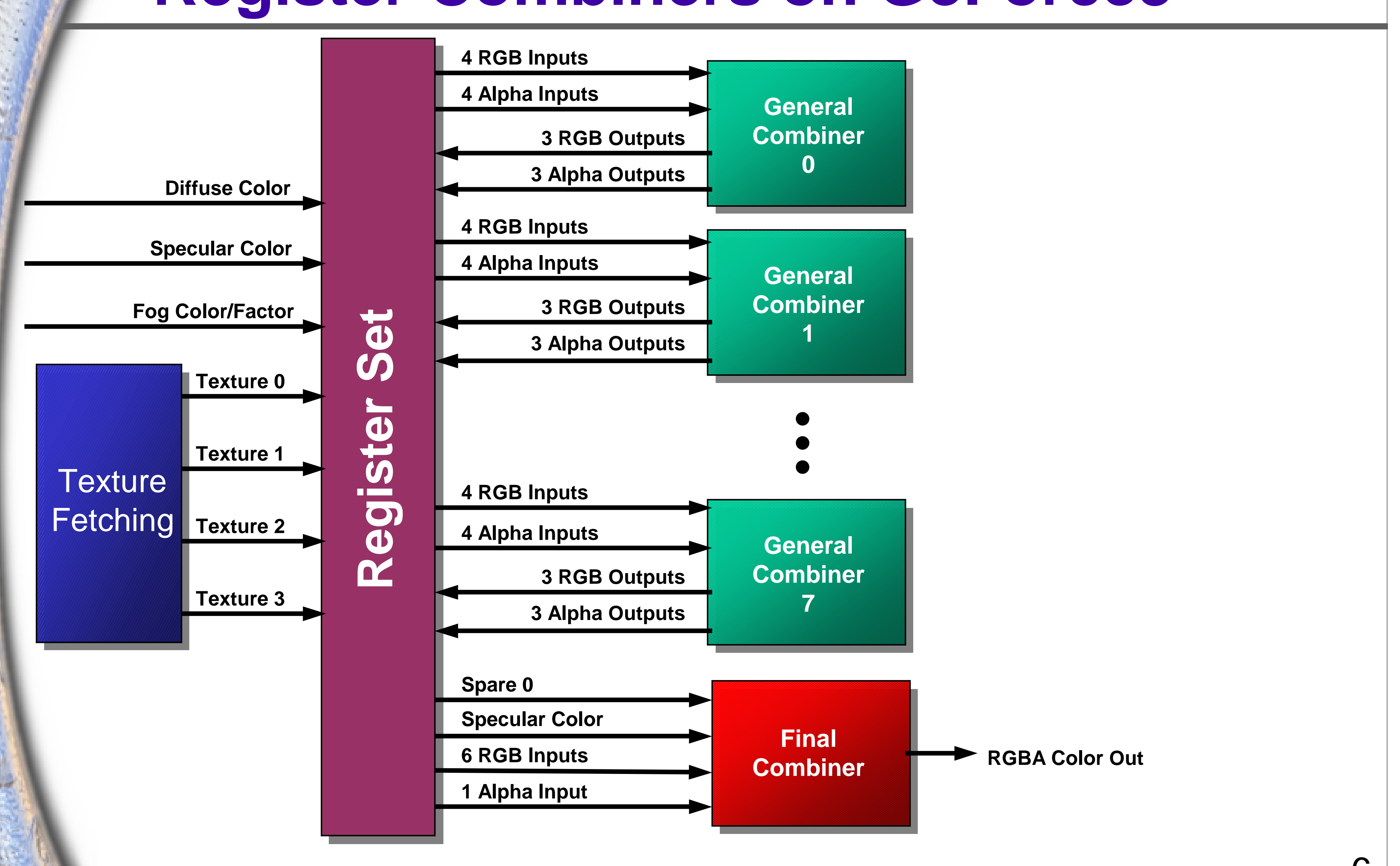
Fog Color/Factor

Fog Application

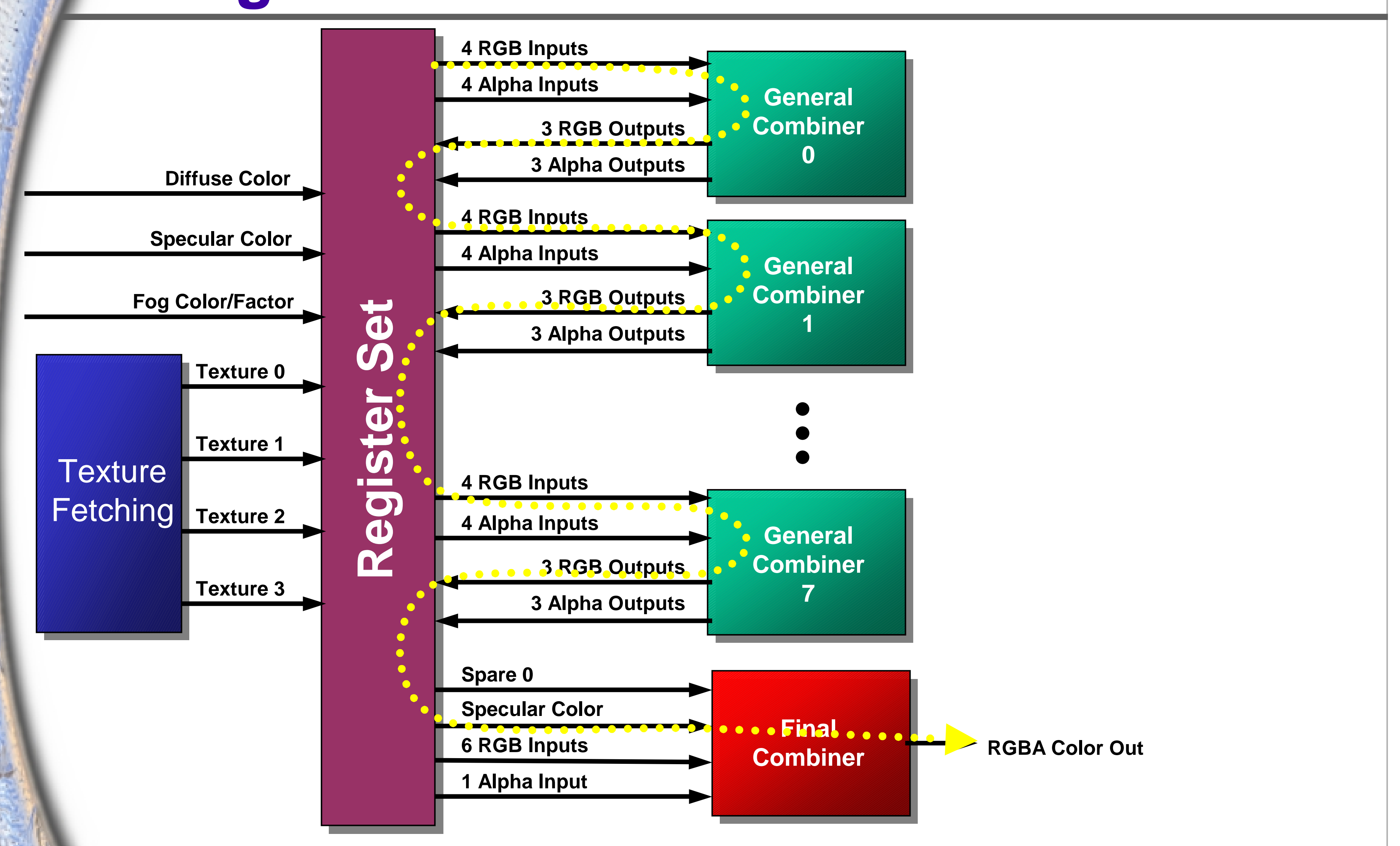
Register Combiners on GeForce2



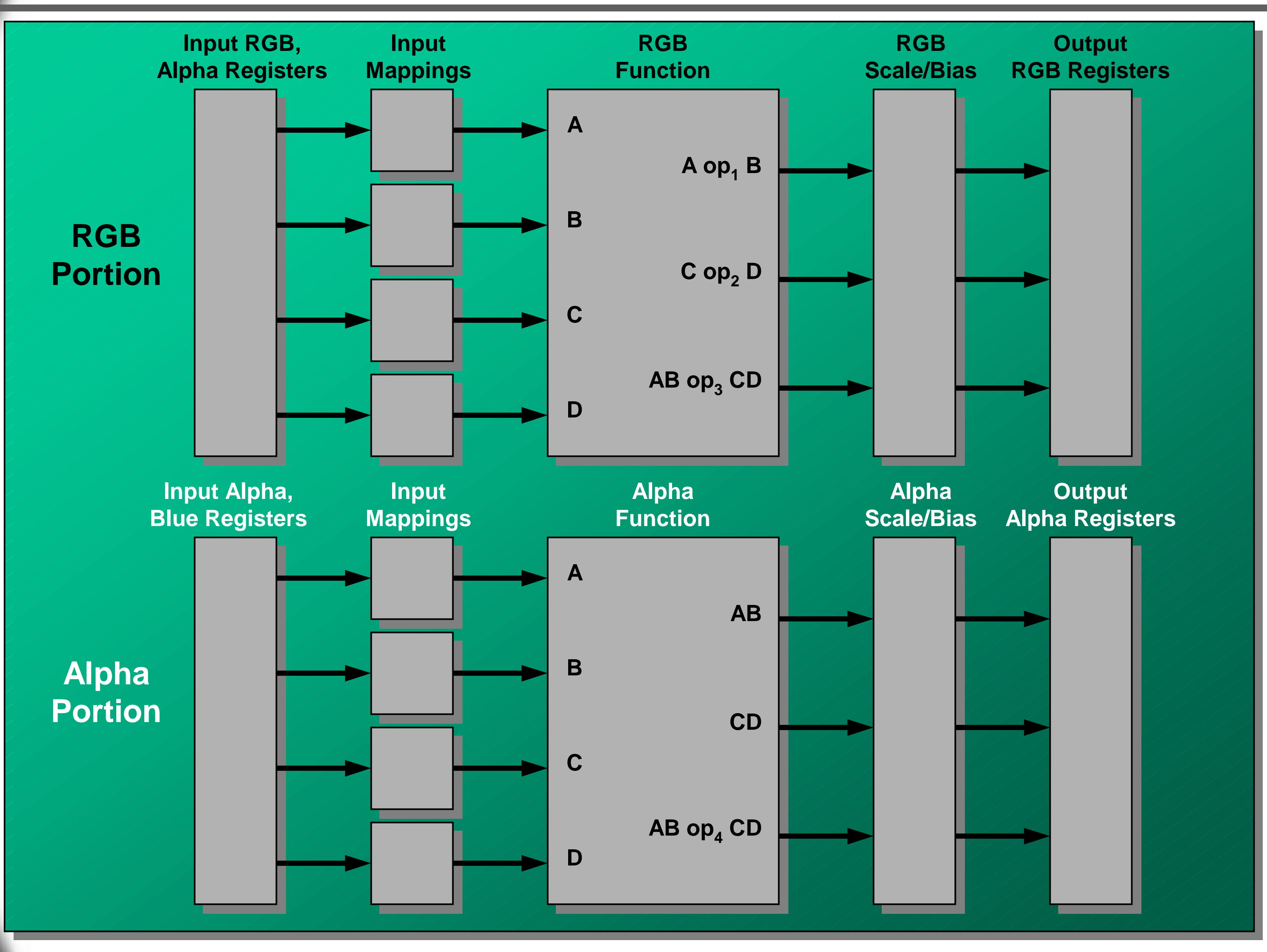
Register Combiners on GeForce3



Register Combiners on GeForce3



The General Combiner

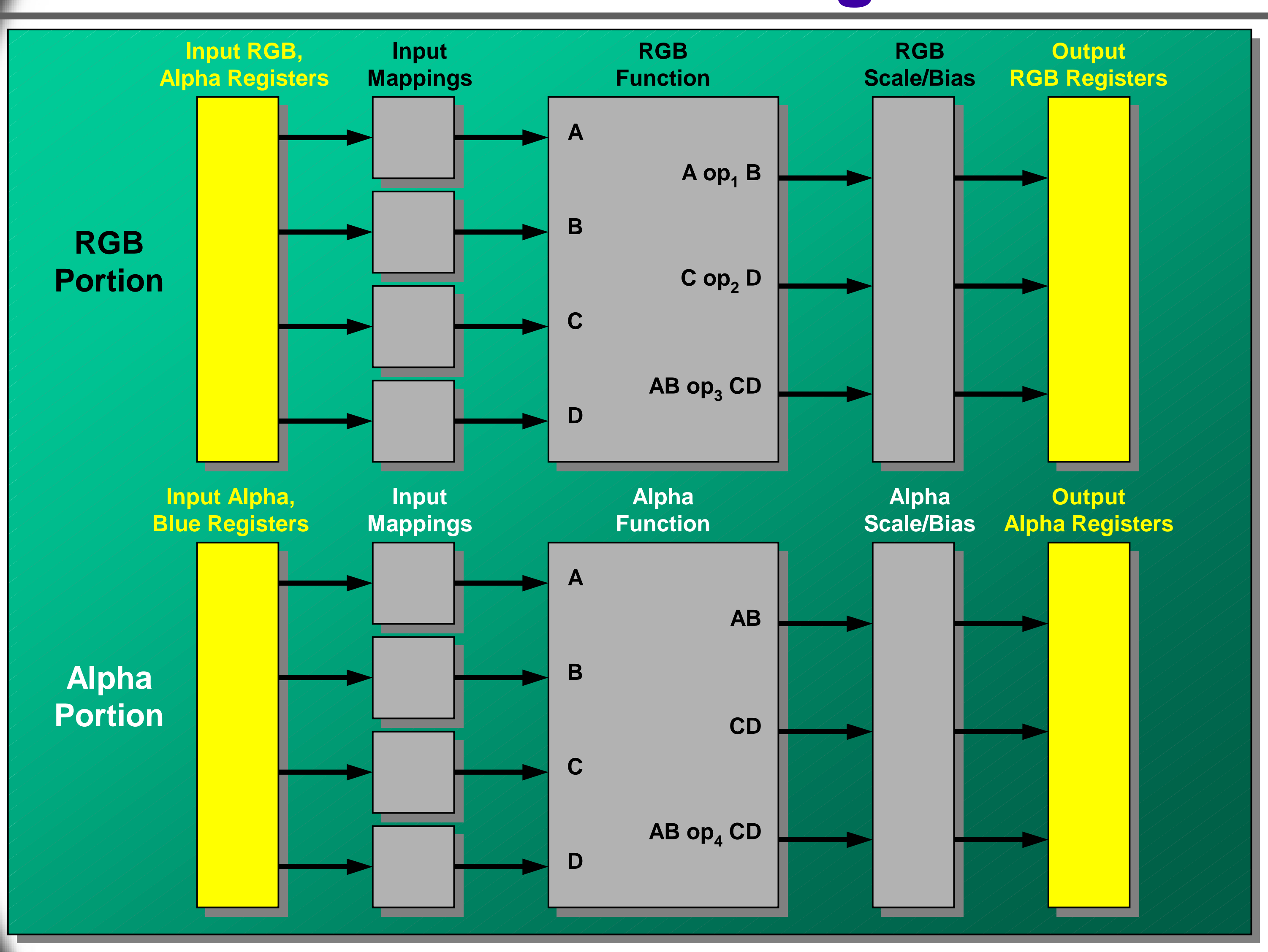


General Combiner Definition

- General combiner is defined within curly brackets
- Each portion (RGB and Alpha) is defined within curly brackets as well
- Example:

```
{    // Beginning of general combiner definition
     rgb
     {        // Beginning of RGB portion definition
              spare0 = col0 * tex0;
     }        // End of RGB portion definition
     alpha
     {        // Beginning of Alpha portion definition
              spare1 = col1 * tex1;
     }        // End of Alpha portion definition
}    // End of general combiner definition
```

General Combiner Register Set



General Combiner Register Set

Register	Name	Read	Write
Diffuse color	`col0`	yes	yes
Specular color	`col1`	yes	yes
Texture 0 color	`tex0`	yes	yes
Texture 1 color	`tex1`	yes	yes
Texture 2 color	`tex2`	yes	yes
Texture 3 color	`tex3`	yes	yes
Spare 0	`spare0`	yes	yes
Spare 1	`spare1`	yes	yes
Constant color 0	`const0`	yes	no
Constant color 1	`const1`	yes	no
Fog color and factor	`fog`	RGB only	no
Zero	`zero`	yes	no
Discard	`discard`	no	yes

Register Usage

Specify a channel, or allow the compiler to infer which one you wanted

Explicit RGB channel:

```
col0.rgb
```

Explicit Alpha channel:

```
col0.a
```

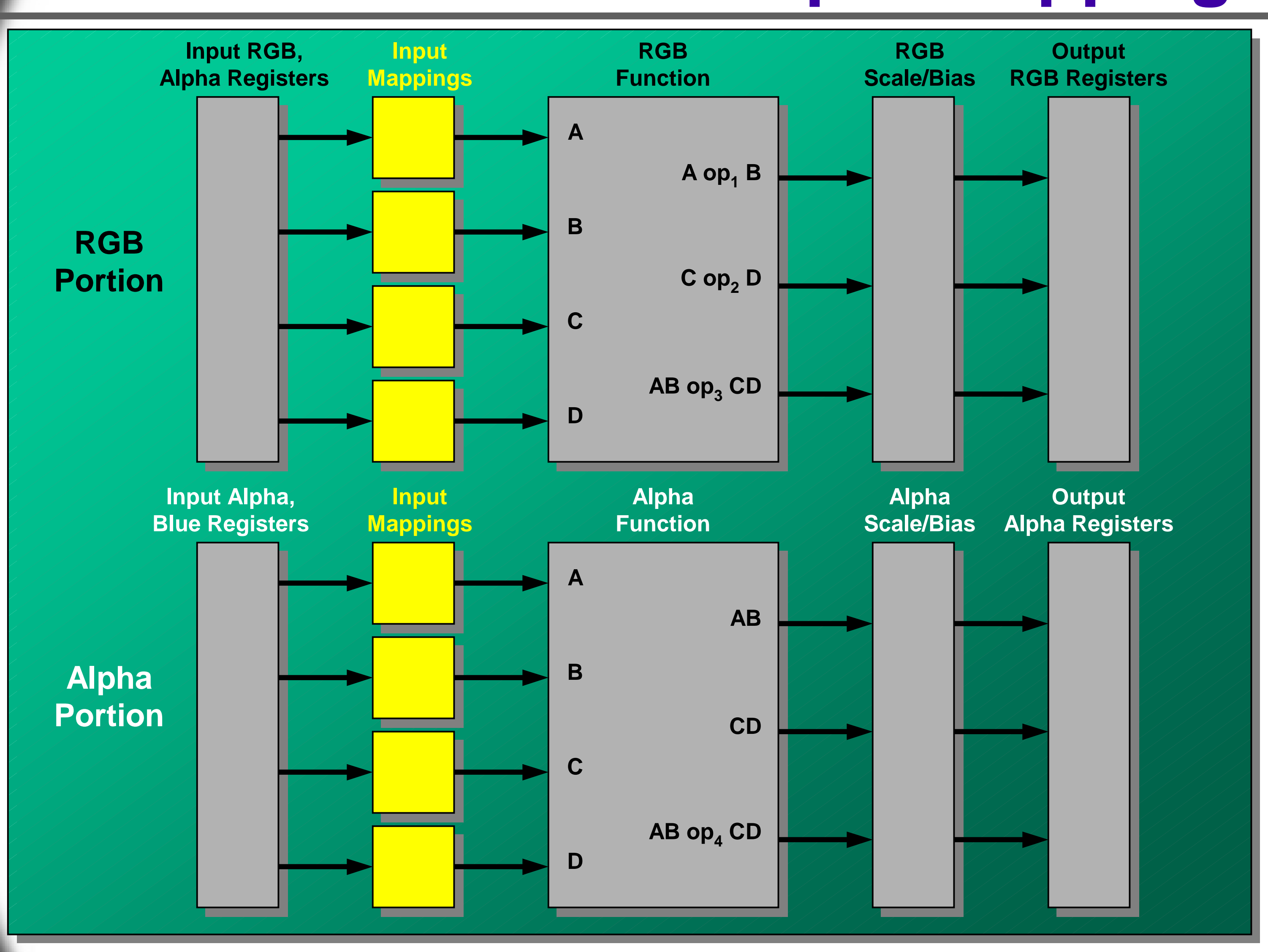
Explicit Blue channel:

```
col0.b
```

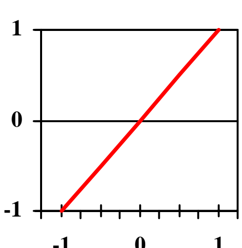
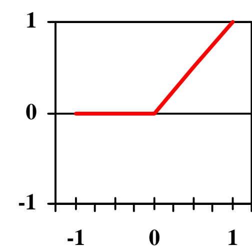
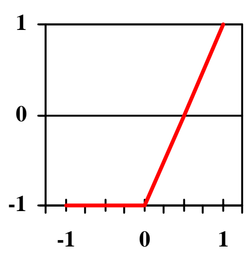
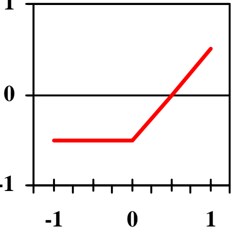
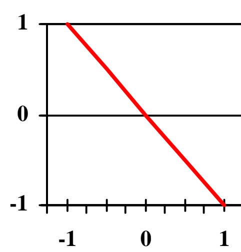
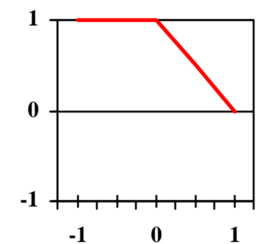
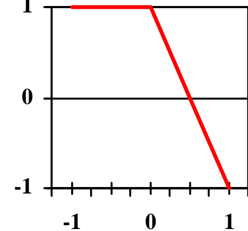
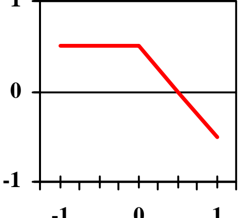
Implicit channel (inherits from portion):

```
col0
```

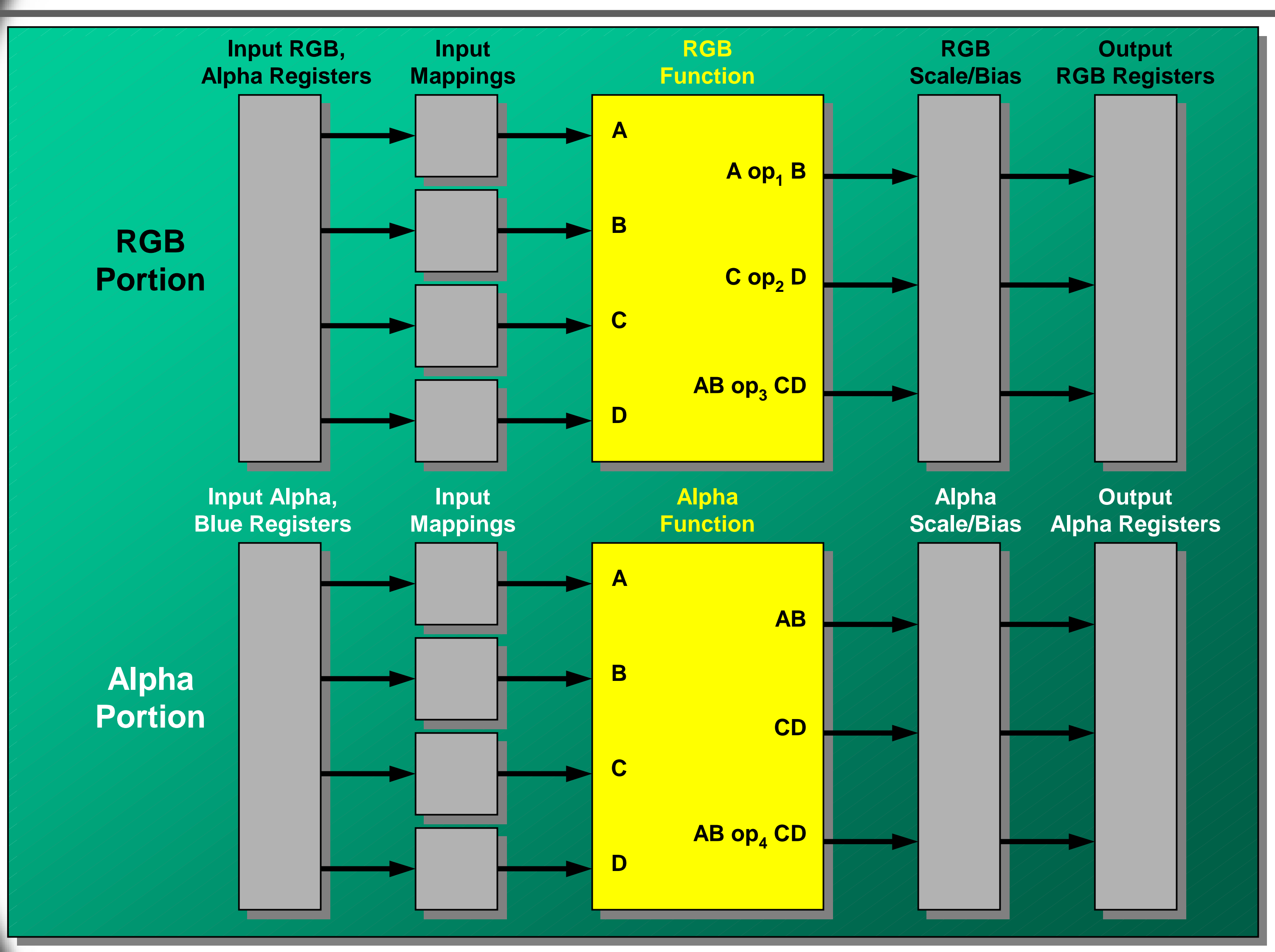
General Combiner Input Mappings



General Combiner Input Mappings

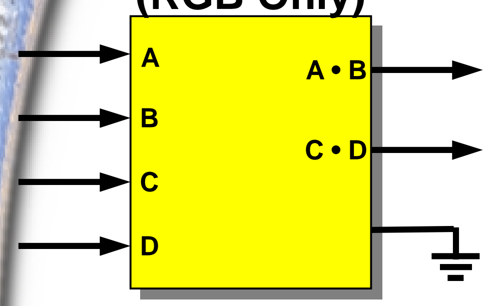
Signed Identity	Unsigned Identity	Expand Normal	Half Bias Normal
f(x) = x	f(x) = max(0, x)	f(x) = 2 * max(0, x) - 1	f(x) = max(0, x) – ½
[-1, 1] → [-1, 1]	[0, 1] → [0, 1]	[0, 1] → [-1, 1]	[0, 1] → [-½, ½]
`tex0`	`unsigned(tex0)`	`expand(tex0)`	`half_bias(tex0)`
			
Signed Negate	**Unsigned Invert**	**Expand Negate**	**Half Bias Negate**
f(x) = -x	f(x) = 1-min(max(0,x),1)	f(x) = -2 * max(0, x) + 1	f(x) = -max(0, x) + ½
[-1, 1] → [1, -1]	[0, 1] → [1, 0]	[0, 1] → [1, -1]	[0, 1] → [½, -½]
`-tex0`	`unsigned_invert(tex0)`	`-expand(tex0)`	`-half_bias(tex0)`
			

General Combiner Functions



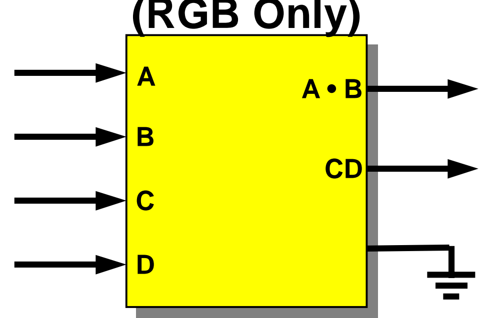
General Combiner Functions

Dot / Dot / Discard
(RGB Only)



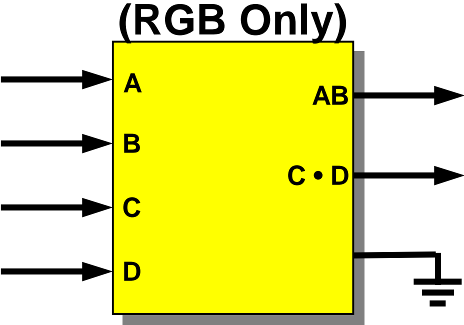

```
spare0 = expand(col0) .
         expand(tex0);
spare1 = expand(col1) .
         expand(tex1);
```

Dot / Mult / Discard
(RGB Only)



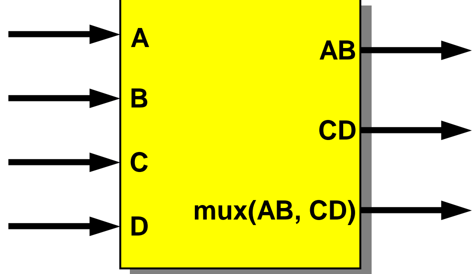

```
spare0 = expand(col0) .
         expand(tex0);
spare1 = col1 * tex1;
```

Mult / Dot / Discard
(RGB Only)




```
spare0 = col0 * tex0;
spare1 = expand(col1) .
         expand(tex1);
```

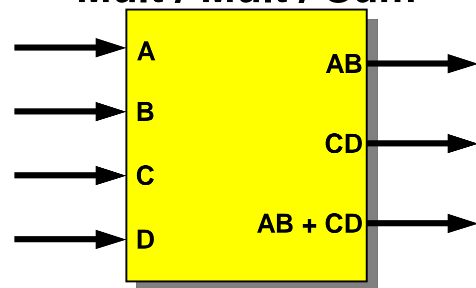
Mult / Mult / Mux



mux(AB, CD) = (Spare0[Alpha] ≤ ½) ? AB : CD

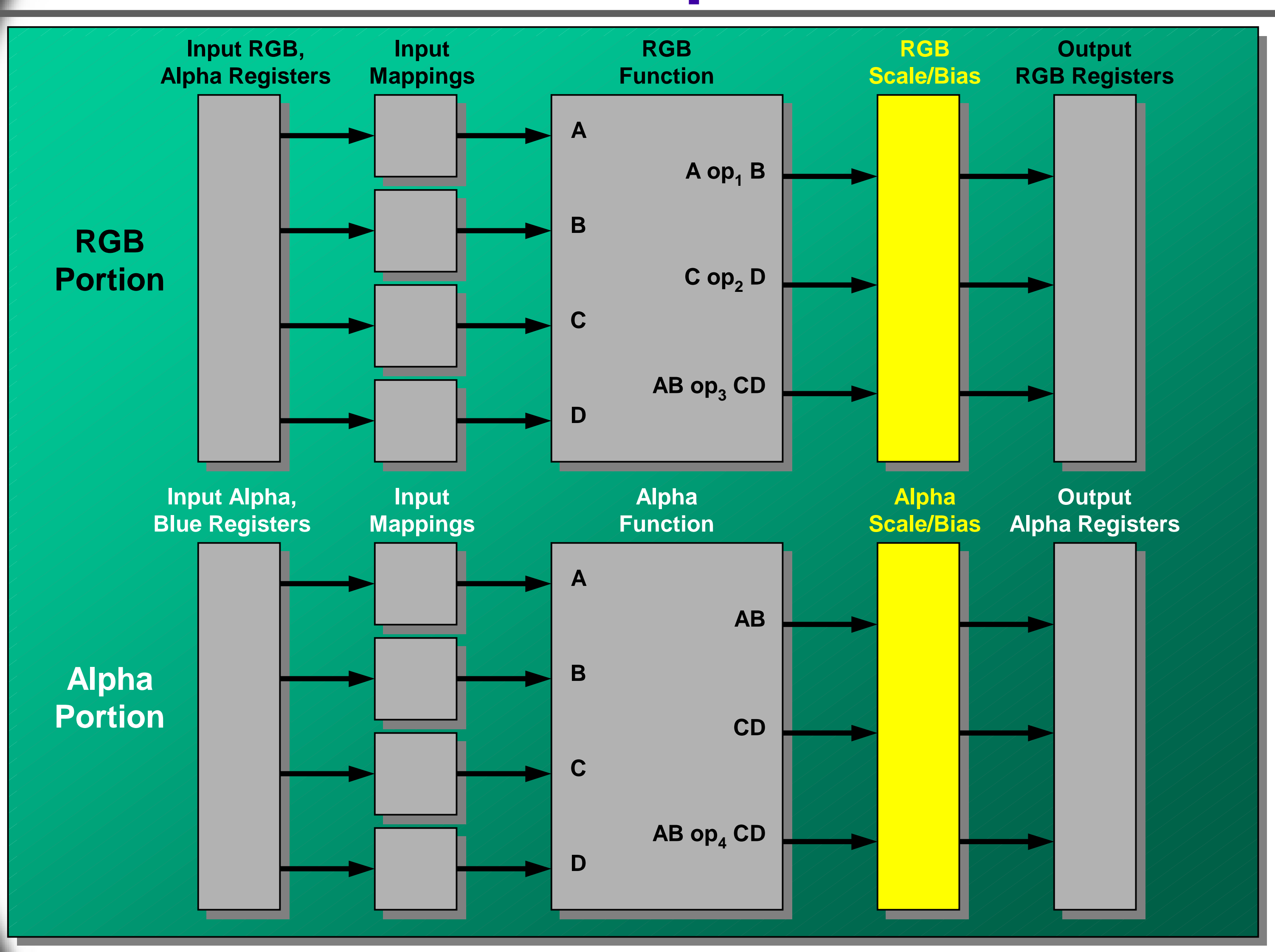
```
discard = col0 * tex0;
discard = col1 * tex1;
spare1  = mux();
```

Mult / Mult / Sum

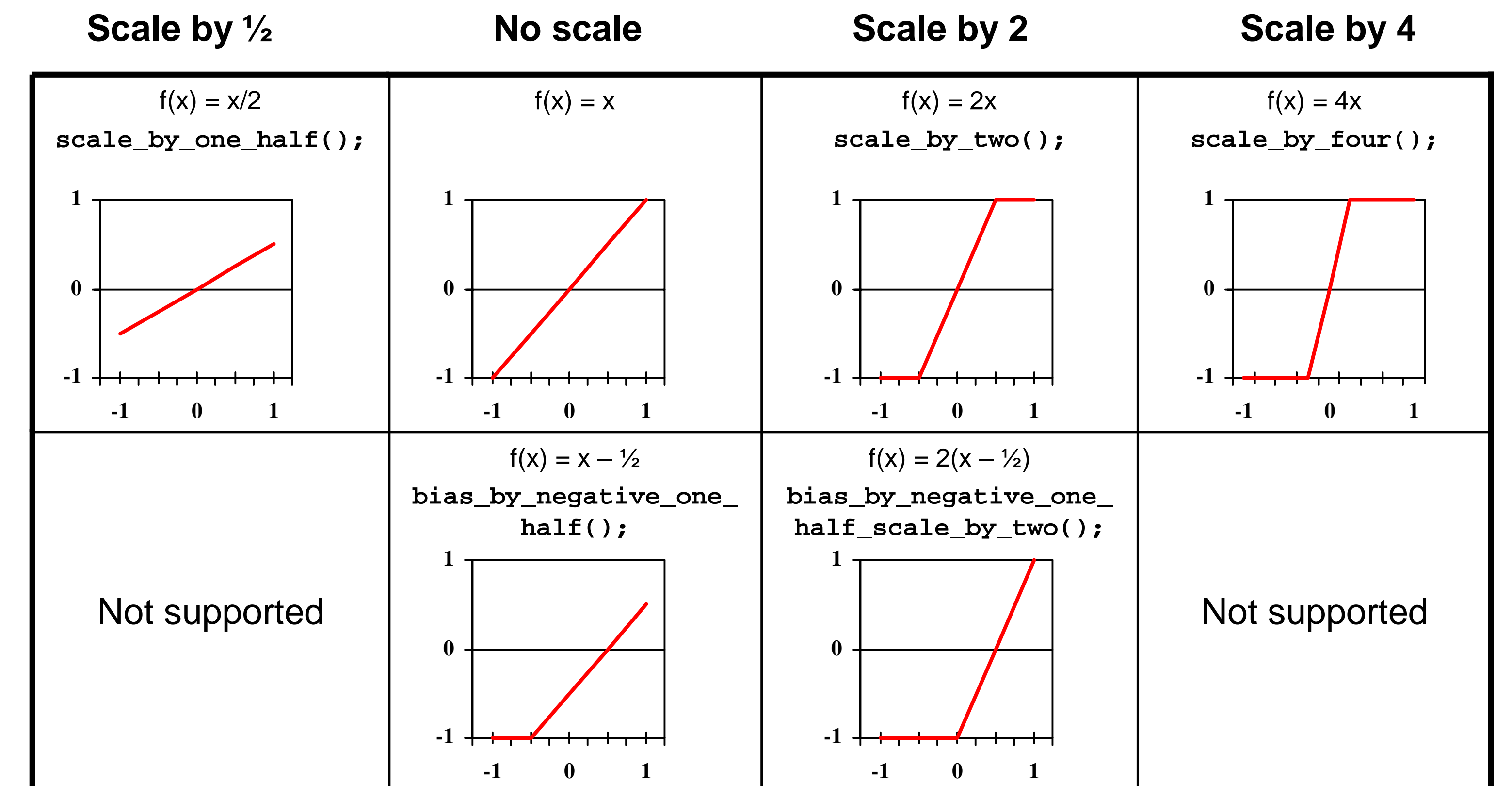



```
discard = col0 * tex0;
spare0  = col1 * tex1;
spare1  = sum();
```

Scale and Bias Options



Scale and Bias Options



Scale and bias operation is defined as:
ClampNegativeOneToOne(Scale * (x + Bias))
OR max(min(Scale * (x + Bias), 1), -1)

Scale and Bias Usage

Scale/bias operation uniformly applied to all outputs

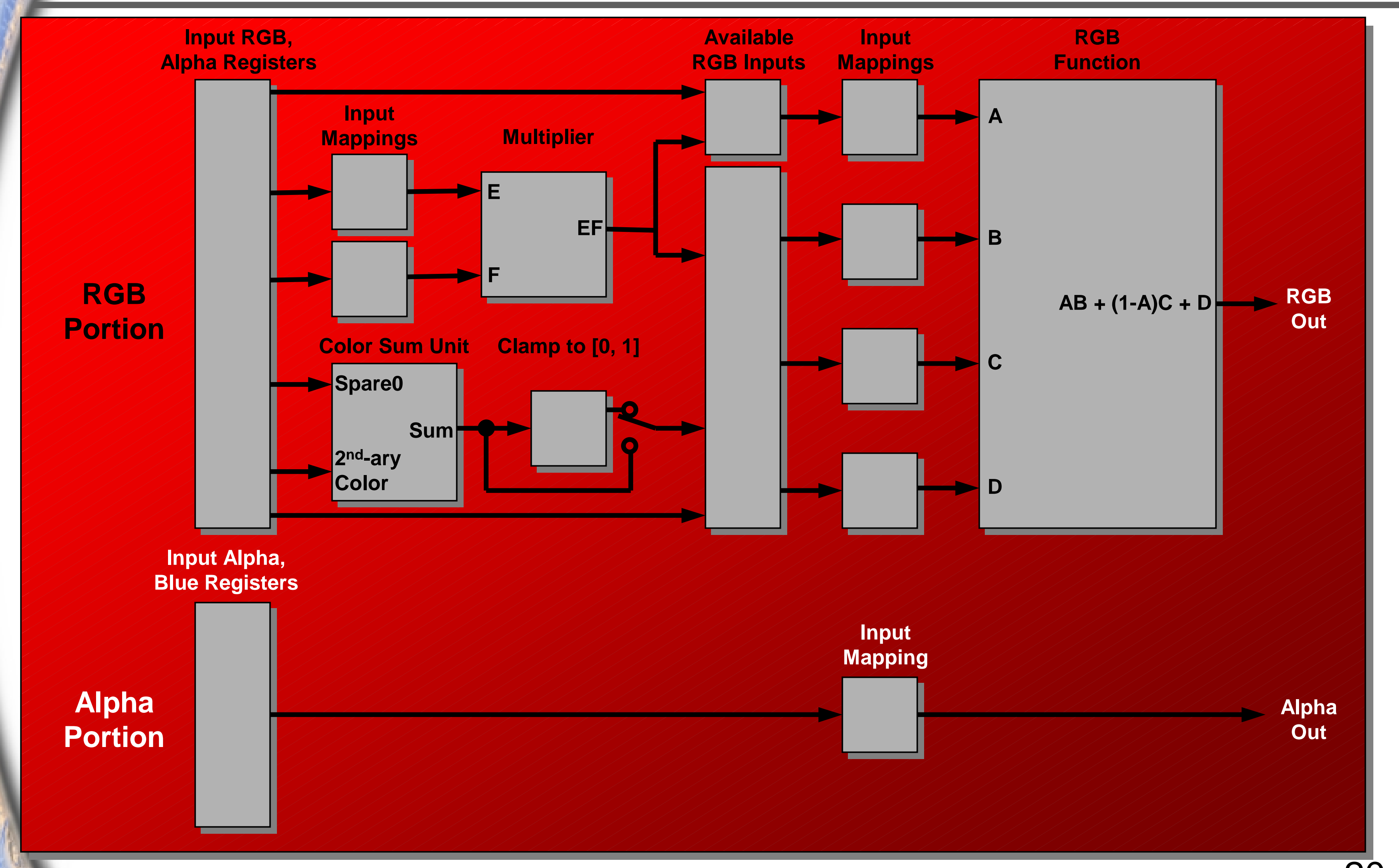
Example A (multiply register value by 8):

```
alpha {
     discard = col0.b;
     discard = col0.b;
     spare0 = sum();
     scale_by_four(); // spare0 = 4 * (col0.b + col0.b)
}
```

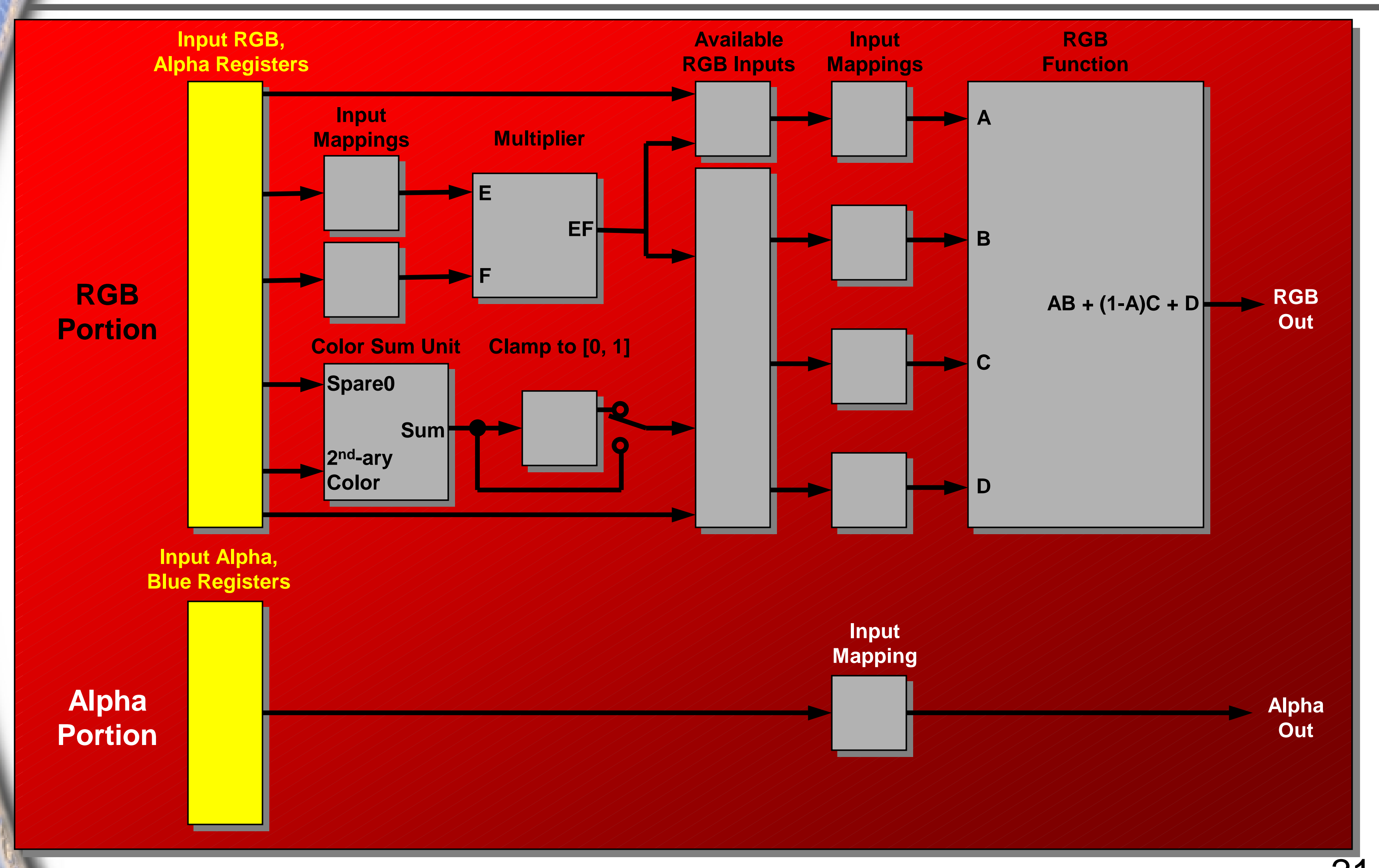
Example B (halve two register values):

```
rgb {
     spare0 = spare0;
     spare1 = spare1;
     scale_by_one_half(); // spare0 /= 2; spare1 /= 2;
}
```

The Final Combiner



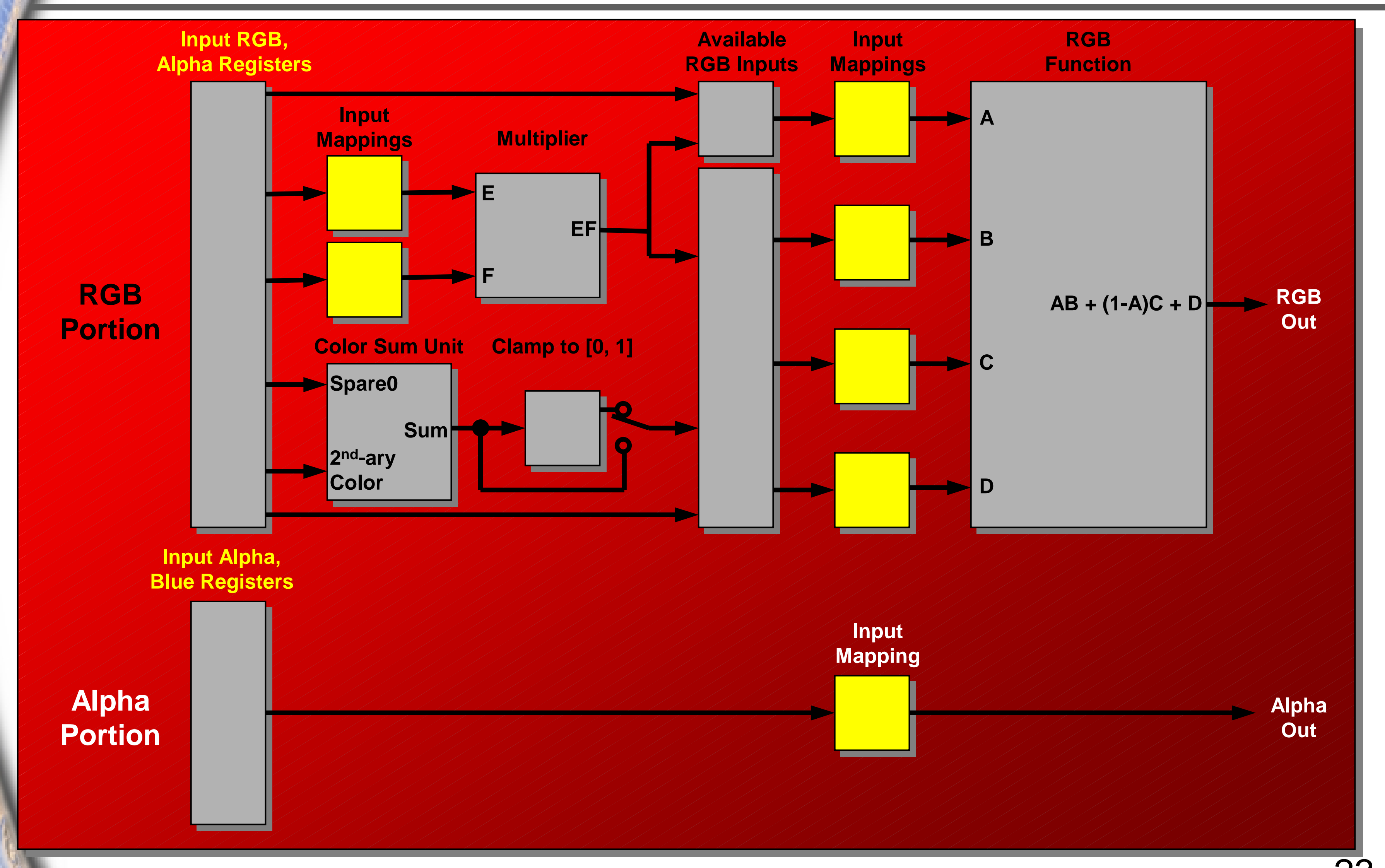
Final Combiner Register Set



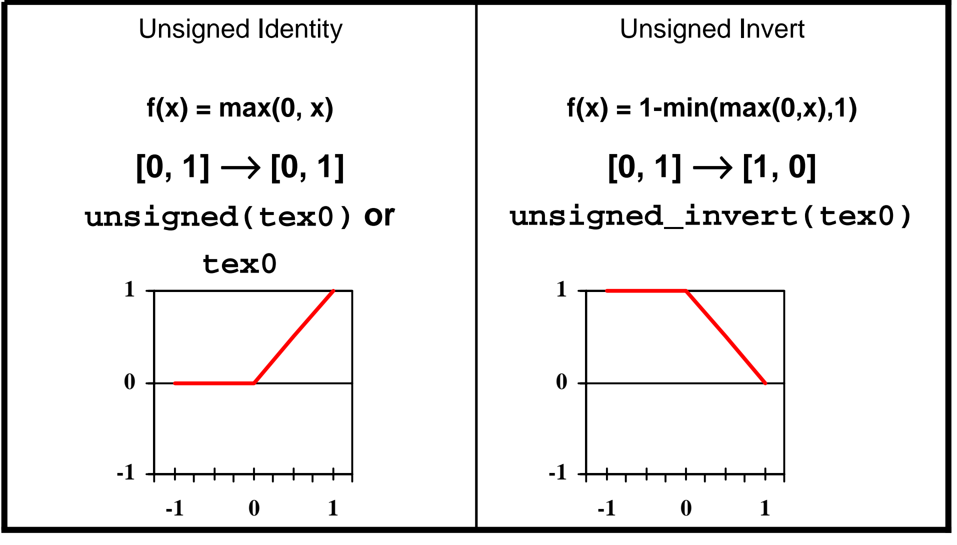
Final Combiner Register Set

Register	Name	Read	Write
Diffuse color	`col0`	yes	no
Specular color	`col1`	yes	no
Texture 0 color	`tex0`	yes	no
Texture 1 color	`tex1`	yes	no
Texture 2 color	`tex2`	yes	no
Texture 3 color	`tex3`	yes	no
Spare 0	`spare0`	yes	no
Spare 1	`spare1`	yes	no
Constant color 0	`const0`	yes	no
Constant color 1	`const1`	yes	no
Fog color and factor	`fog`	yes	no
Zero	`zero`	yes	no

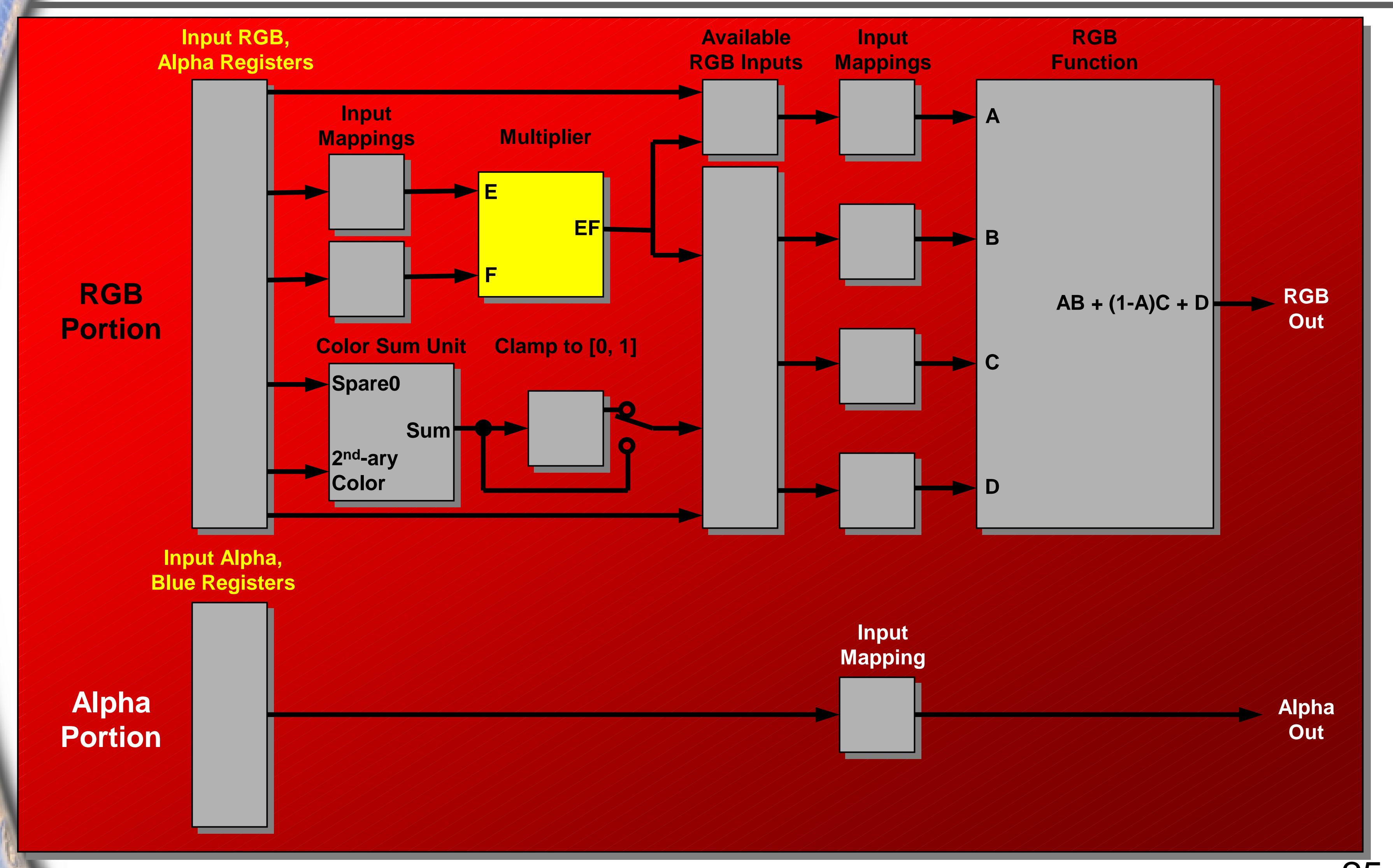
Final Combiner Input Mappings



Final Combiner Input Mappings



Final Product



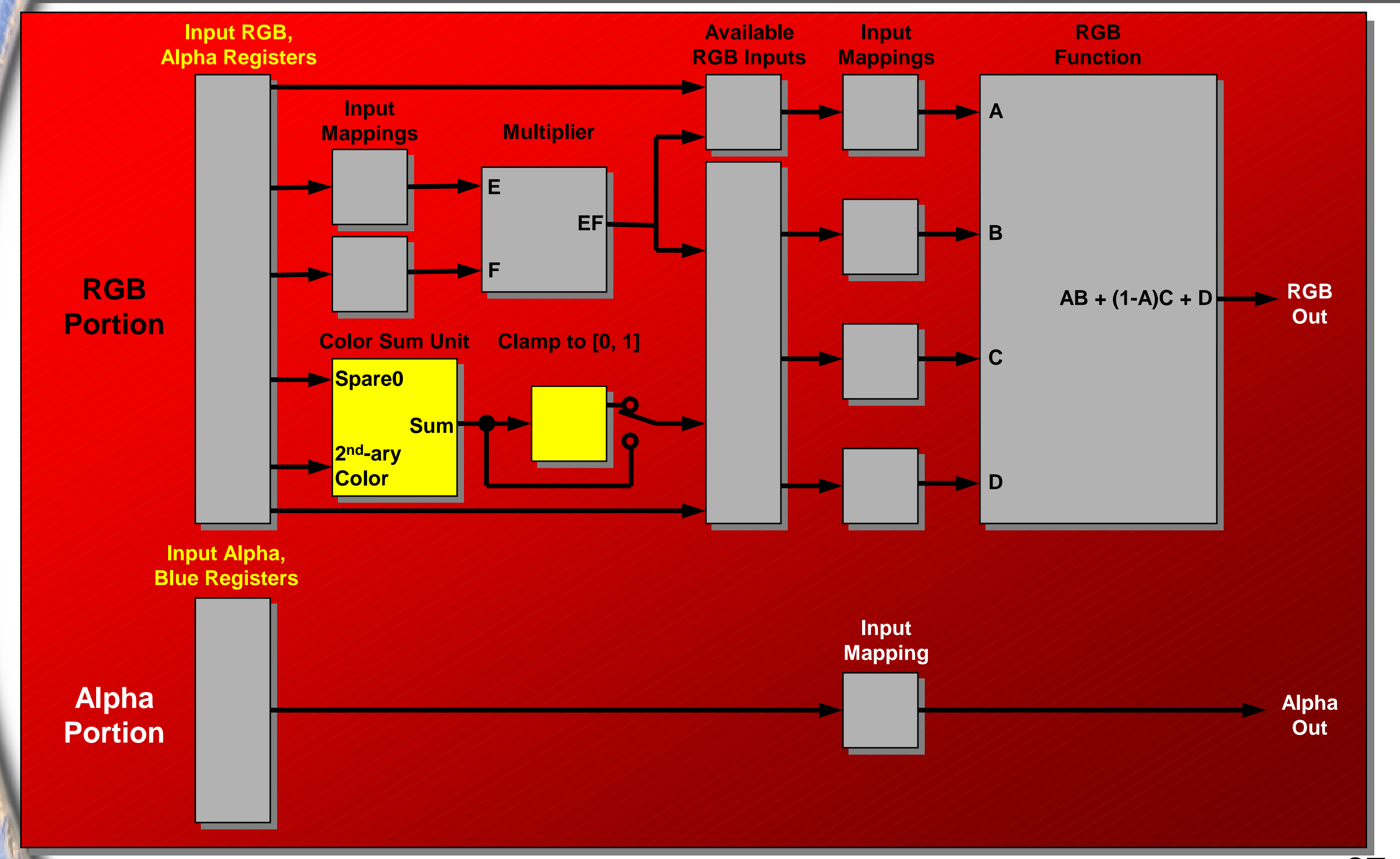
Final Product

One may use the final product to multiply any two registers together, and then use the result in the final computation:

```
final_product = tex0 * unsigned(col0);
```

Computing the final_product is optional, but must be placed after the last combiner definition, but before the final assignment to `out`

Final Color Sum



Final Color Sum

The color_sum register is hard-coded to compute:

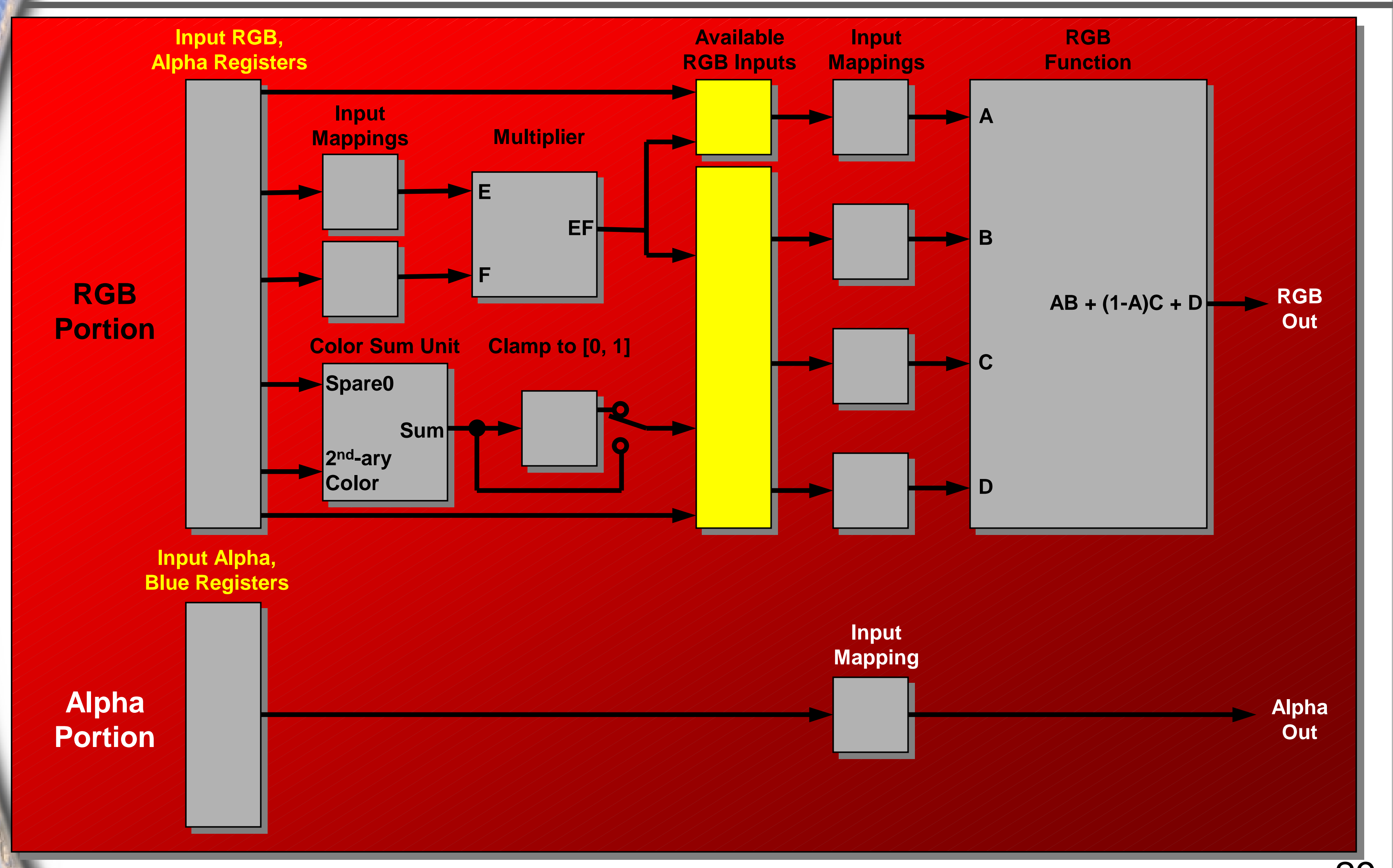
spare0.rgb + col1.rgb

By default, the value is unclamped and ranges [0,2]

If one would like this value clamped [0,1], after the last general combiner definition, call:

```
clamp_color_sum();
```

Available RGB Inputs

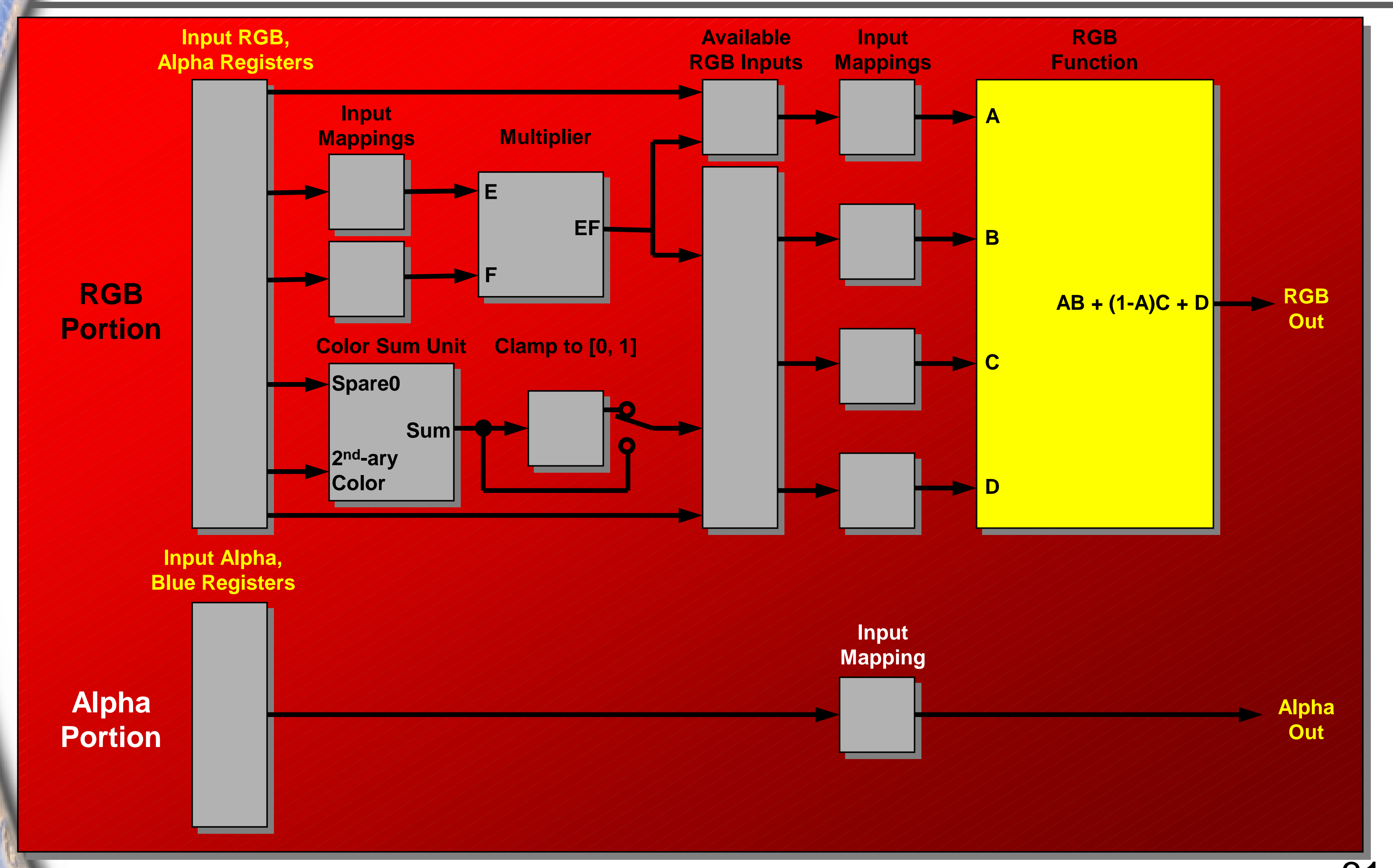


Available RGB Inputs

In addition to the register listed before, the RGB inputs to the final combiner may also be:

- `final_product` (final product result)
- `color_sum` (potentially clamped color sum result)

Final Combiner Function



Final Combiner Function

The final combiner RGB function is hard-coded to compute:

A * B + (1 – A) * C + D

for any given registers A, B, C and D. The only restriction is that A may not be `color_sum`

In nvparse, the RGB final equation can take many forms

Final Combiner RGB Function Forms

Trivial assignment:

```
out.rgb = tex0;
```

Product:

```
out.rgb = tex0 * final_product;
```

Sum:

```
out.rgb = tex0 + final_product;
```

Linear interpolation (A*B + (1-A)*C):

```
out.rgb = lerp(fog.a, const0, color_sum);
```

Linear interpolation and sum:

```
out.rgb = lerp(fog.a, const0, color_sum) + const1;
```

Final Combiner Alpha Function Forms

By comparison, the alpha final equation is restricted to a mere assignment:

```
out.a = unsigned_invert(zero); // set to 1

out.a = tex0;

out.a = col1.b;
```

Currently, both `out.rgb` and `out.a` must be specified, but this will likely be changed in a future version

User Defined Constant Colors

Two constant colors can be defined on GeForce

Colors are 4 elements (RGBA) in range [0,1]

GeForce3 optionally has more constants available

- **Two constants for the final combiner**
- **Two unique constants for *each* general combiner**
- **Total of 18 constants!**

Constants defined as:

```
const0 = ( float, float, float, float );
const1 = ( float, float, float, float );
```

Follows C-style scoping

User Defined Constant Colors

Scoping example:

```
const0 = (0, 0, 0, 0); // global color
{
   // this assignment applies only to combiner 0
   const0 = (1, 1, 1, 1);
   rgb
   {
       spare0 = const0;
   }
}
out.rgb = spare0;   // (1, 1, 1)
out.a   = const0.a; // 0
```

Invoking the Parser

nvparse is a DLL with only two entry points:

```
void nvparse(const char * input_string);
```

where `inputString` points to the program

```
char * const * const nvparse_get_errors();
```

will return list of parser errors, or null pointer if none found

Register combiner programs must start with:

```
!!RC1.0
```

Newlines at the end of each line of the program will ease in debugging as parser supplies line numbers for errors when found

Example

```
nvparse(
"!!RC1.0\n"
{\n"
"  rgb {\n"
"       spare0 = expand(col0) . expand(tex1);\n"
"       spare1 = expand(col1) . expand(tex1);\n"
"  }\n"
"}\n"
"final_product = spare1 * spare1;\n"
"out.rgb = spare0 * tex0 + final_product;\n"
"out.a = unsigned_invert(zero);\n"
);

for (const char** errors= nvparse_get_errors();
*errors;
errors++)
       fprintf(stderr, *errors);

glEnable(GL_REGISTER_COMBINERS_NV); // Don't forget this!
```

Questions?

Send to jspitzer@nvidia.com