GPU

This document is a translation from Spanish to English of some parts of the graduate thesis “Diseño e Implementación de una Unidad de Procesamiento Gráfico Implementada en FPGA” elaborated by Diego Andrés González Idárraga and directed by the teacher Luis Miguel Capacho Valbuena for the Electronic Engineering Program of the University of Quindío.

2015

1 Introduction

The graphics processed by the GPU are defined as a set of vertices that contain spatial information, i.e. vectors with coordinates $[x \quad y \quad z]$ in three-dimensional Cartesian space, and additional information of color or texture coordinates.

The processing that is performed has 4 phases:

1. A group of vertices is processed as a point list, a line list, a line strip, a triangle list, a triangle strip or a triangle fan.
2. The given vertices are transformed three-dimensionally based on a 4x4 transformation matrix, being able to translate them, rotate them, scale them and/or project them.
3. Depending on whether the vertices correspond to a set of points, lines or triangles, the visible region of these figures is calculated by cutting the parts that are in front of the near view plane, and the parts that are behind the far plane of vision.
4. The information contained in the vertices is mapped in the area defined by these, at this stage it is assumed that the vertices already been projected from three-dimensional space to a space bidimencional.

The area mentioned in step 4 of processing corresponds to a section of memory that can be read and then sent to a screen for viewing.

The graphics processing unit is controlled by the Nios II processor, and is embedded in a system on a single chip along with other hardware modules, such as the memory controller, keyboard and mouse controller, among others.

2 Hardware Modules

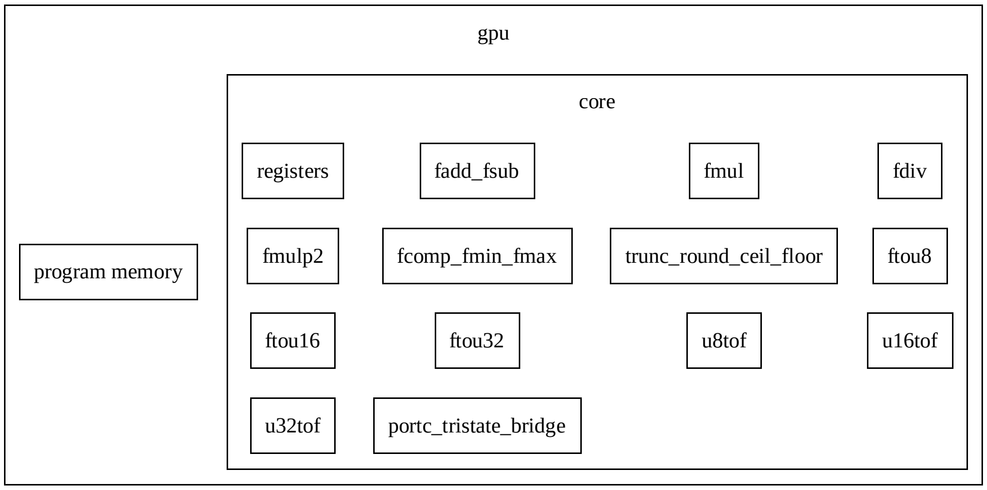


Figure 1. Block diagram of the overall structure of the system.

The hardware design of the graphics processing unit consists of a series of modules organized into various groups, in addition, there was also the need to design a module for communication ps/2 because in the version of Quartus II 11.1sp2 that used during labor had no available module for communication with peripherals such as the keyboard and mouse:

2.1 pfloat (pipelined float)

Hardware modules in this group are responsible for performing various operations on floating point numbers in single precision with a programmable latency to certain values that can be exploited to perform operations on a segmented and thus increase the rate at which they can be implemented.

Table 1. Format float structure.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
sign	**exponent**								**fraction**																						
1	8								23																						

```
type pfloat is record
    sign     : std_logic;
    exponent : unsigned(7 downto 0);
    fraction : unsigned(22 downto 0);
end record;
```

```
function to_pfloat(x : std_logic_vector(31 downto 0)) return pfloat is
begin
    return (x(31), unsigned(x(30 downto 23)), unsigned(x(22 downto 0)));
end function;

function to_stdlogicvector(x : pfloat) return std_logic_vector is
begin
    return x.sign&std_logic_vector(x.exponent&x.fraction);
end function;
```

The pfloat type defined in pfloat_pkg.vhd file as shown in the above code, and is structured according to the IEEE 754 standard for the binary format for single precision floating point.

Additionally these modules have the generic variable **`USE_SUBNORMAL`** allows you to program if the modules use the range of numbers subnormal defined in the IEEE 754 standard, this in order that in case of not using that range you can save a significant amount of hardware implementation.

As shown in equation (1) the range corresponds to subnormal numbers very close to zero.

$$f_{\text{normal}} = (-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times 1.\text{fraction} \quad (1)$$

$$f_{\text{subnormal}} = (-1)^{\text{sign}} \times 2^{-126} \times 0.\text{fraction} \quad (2)$$

In the above file three special constants that are useful for other functions and modules are also defined:

```
        constant ZERO     : pfloat := ('0', "00000000", "00000000000000000000000");
        constant INFINITY : pfloat := ('0', "11111111", "00000000000000000000000");
        constant NAN      : pfloat := ('0', "11111111", "10000000000000000000000");
```

The numbers in single precision floating point are classified into 5 groups: zero, subnormal, normal, infinite, and NaN. To know which of the 5 groups belong a number in floating point single precision the **`to_pfloat_c`** function is designed, which receives as its first argument the number to be rated as second argument a boolean that determines whether you want to use the range of subnormal numbers:

```
function to_pfloat_c(
    x : pfloat;
    USE_SUBNORMAL : boolean
) return pfloat_c is
    variable x_c : pfloat_c;
begin
    x_c.exponent_or_reduce  := or_reduce(x.exponent);
    x_c.exponent_and_reduce := and_reduce(x.exponent);
    x_c.fraction_or_reduce  := or_reduce(x.fraction);

    if USE_SUBNORMAL then
        x_c.zero      := (x_c.exponent_or_reduce = '0') and (x_c.fraction_or_reduce = '0');
        x_c.subnormal := (x_c.exponent_or_reduce = '0') and (x_c.fraction_or_reduce = '1');
    else
        x_c.zero      := x_c.exponent_or_reduce = '0';
        x_c.subnormal := false;
    end if;
    x_c.normal        := (x_c.exponent_or_reduce = '1') and (x_c.exponent_and_reduce = '0');
    x_c.infinity      := (x_c.exponent_and_reduce = '1') and (x_c.fraction_or_reduce = '0');
    x_c.nan           := (x_c.exponent_and_reduce = '1') and (x_c.fraction_or_reduce = '1');

    return x_c;
end function;
```

If the argument is false **`USE_SUBNORMAL`** all numbers within the range subnormal numbers are treated as if they were exactly zero.

In Figure 2 the hardware resulting from applying the **`to_pfloat_c`** function using subnormal numbers range is observed, and in Figure 3 the hardware resulting from applying the same function without using said range is observed.

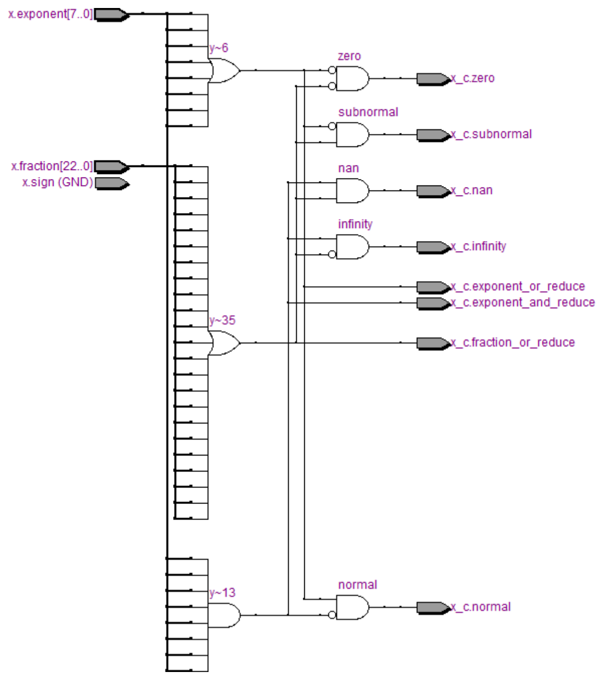


Figure 2. **`x_c <= to_pfloat_c(x, true);`**

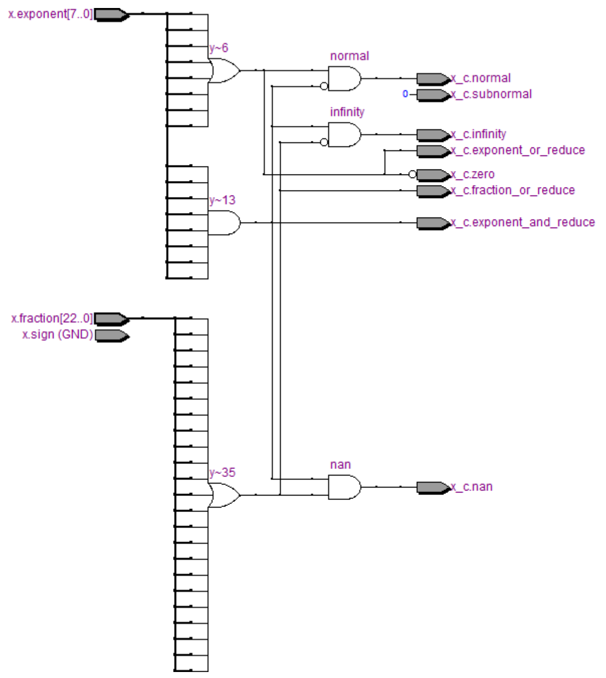


Figure 3. **`x_c <= to_pfloat_c(x, false);`**

To convert a number in **`unsigned`** format of any size to the format in single precision floating point function called **`to_pfloat`** was designed, which receives as argument the unsigned integer to be converted and optionally the number of subdivisions that are to be done in the calculation of leftward.

The **SUBDIVIDE** argument specifies the number of subdivisions does not affect in any way the result of the conversion rate, but decreases the critical path of the resulting hardware increasing the amount of hardware required to perform the operation slightly.

```
function to_pfloat(
    x         : unsigned;
    SUBDIVIDE : positive := 1
) return pfloat is
    variable shift    : shift_t;
    variable exponent : natural;
    variable x_1      : unsigned(x'length-1 downto 0);
    variable fraction : unsigned(23 downto 0);
    variable y        : pfloat;
begin
    shift := shift_calc(x, SUBDIVIDE);
    exponent := x'length+126-shift.i;

    if not(shift.valid) then
        y := ZERO;
    elsif (x'length+126 > 254) and (exponent > 254) then
        y := INFINITY;
    else
        x_1 := shift_left(x, shift.i);
        fraction := (others=> '0');
        fraction(23 downto maximum(24-x'length, 0)) := x_1(x'length-1 downto maximum(x'length-24, 0));
        y := ('0', to_unsigned(exponent, 8), fraction(22 downto 0));
    end if;

    return y;
end function;
```

Significantly checking **`(x'length+126 > 254)`** can be solved in "compile time" and serves to save hardware when the conversion rate is from unsigned integers less than or equal to 128 bits in size, which could never generate an overflow into infinity as a result.

In Figure 4 the resulting hardware **`to_pfloat`** use the function to convert an integer unsigned 8-bit format was in single precision floating point using a single subdivision in the calculation of displacement is observed.

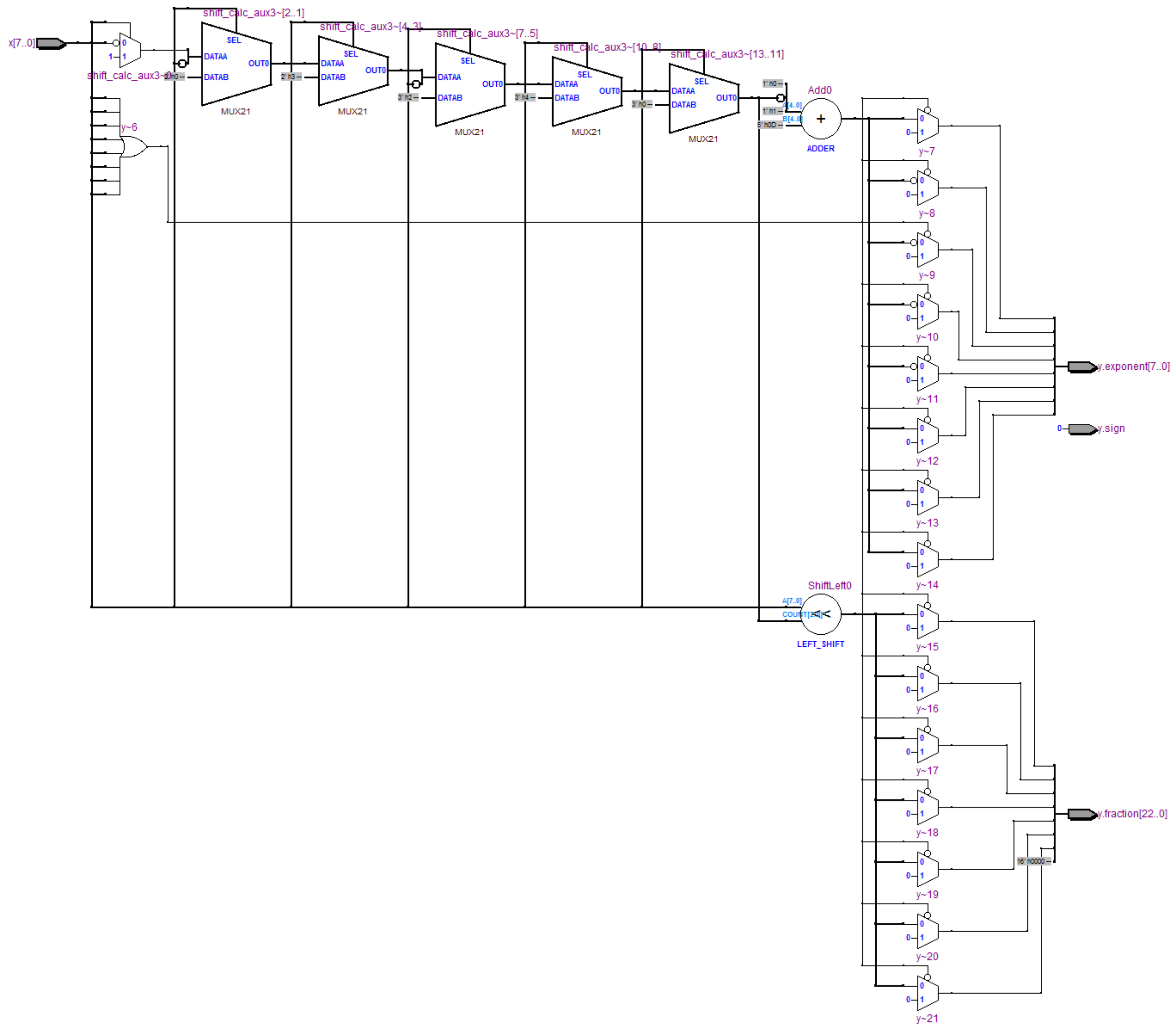


Figure 4. **`y <= to_pfloat(x);`**

Similarly, the **`to_unsigned`** function was also designed to convert a floating point number to the single-precision unsigned integer format sizes.

```
function to_unsigned(
    x    : pfloat;
    size : natural
) return unsigned is
    variable x_c      : pfloat_c;
    variable bias     : natural;
    variable y        : unsigned(size-1 downto 0);
    variable fraction : unsigned(23 downto 0);
begin
    x_c := to_pfloat_c(x, false);
    bias := size+126;

    if x_c.nan or (x.sign = '1') then
        y := (others=> '0');
    elsif x_c.infinity or ((bias < 254) and (bias < x.exponent)) then
        y := (others=> '1');
    else
        fraction := '1'&x.fraction;
        y := (others=> '0');
        y(size-1 downto maximum(size-24, 0)) := fraction(23 downto maximum(24-size, 0));
        y := shift_right(y, to_integer(bias-x.exponent));
    end if;
```

```
        return y;
    end function;
```

Significantly verification **(bias < 254)** can be solved in "compile time" and serves to save hardware when converting to integer type is performed unsigned greater than or equal to 128 bits in size, which only overflow when the number of single precision floating point is infinite.

In Figure 5 the output using hardware **to_unsigned** function for converting a floating point number of single precision made in unsigned integer format by 8 bits is observed.

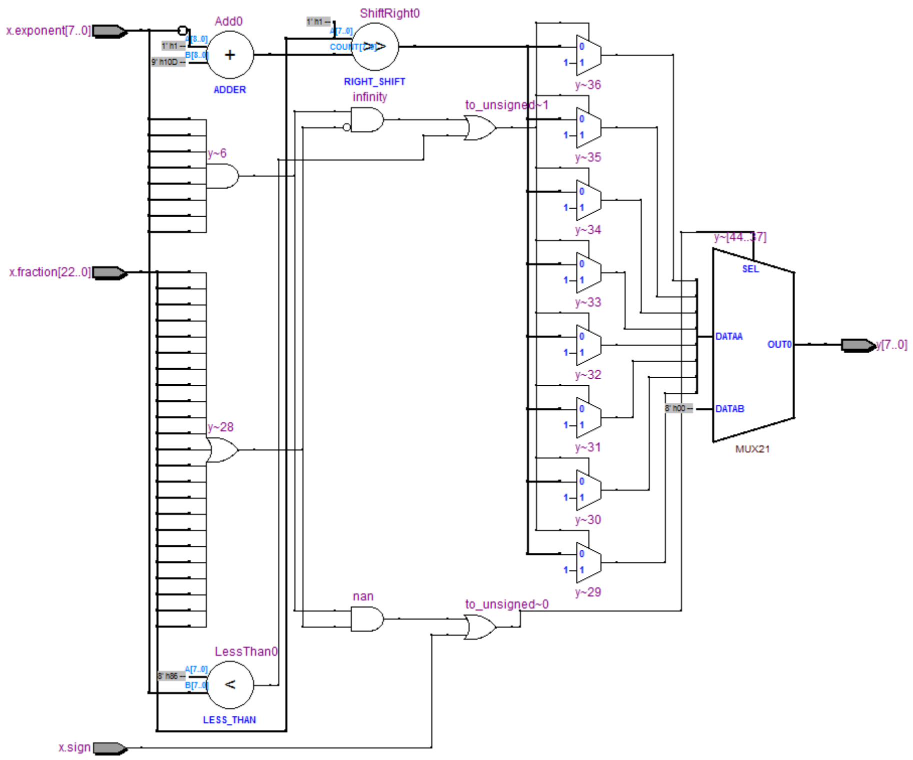


Figure 5. **y <= to_unsigned(x, 8);**

2.1.1 fadd_fsub

Module responsible for the operation of addition or subtraction of two numbers in single precision floating point.

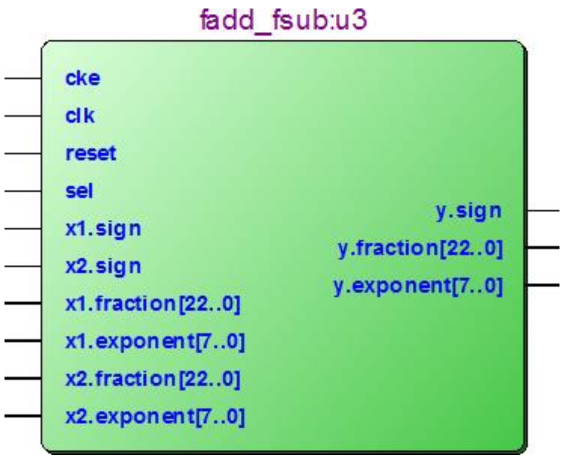


Figure 6. RTL Viewer's fadd_fsub external module.

```
entity fadd_fsub is
    generic(
        USE_SUBNORMAL  : boolean;
        ROUND_STYLE    : float_round_style;
        LATENCY        : natural
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        sel  : in std_logic;
        x1   : in pfloat;
        x2   : in pfloat;

        y : out pfloat
    );
end entity;
```

The **sel** input port determines which of the two operations will be carried out:

Table 2. Selection module operation fadd_fsub.

sel	Operation
'0'	Addition: y = x1+x2
'1'	Subtraction: y = x1-x2

Table 3 shows the operations performed internally by the module and depend on the use or not of the subnormal range are described.

Table 3. Internal Operations module fadd_fsub.

Operations	With subnormal range	Without subnormal range
Determining which of the magnitudes of the two numbers is larger or not to perform an exchange between the numbers	X	X
Rightward shift of the mantissa of the number with smaller magnitude	X	X
Add or subtract the mantissa of the number with the largest magnitude and shifted mantissa normalization led to the result with the exponent	X	X
Dependent offset correction exponent range denormalized	X	
Rounding-normalized result	X	X
Special cases	X	X

Special cases mentioned in Table 3 are further described in Table 4.

Table 4. Special cases of fadd_fsub module.

Operation	x1	x2	Description	Result
Addition or Subtraction	NaN	Any value	$\text{NaN} \pm x$	NaN
	Any value	NaN	$x \pm \text{NaN}$	
Addition	$\pm\infty$	$\mp\infty$	$\pm\infty + \mp\infty$	
Subtraction	$\pm\infty$	$\pm\infty$	$\pm\infty - \pm\infty$	
Addition	$\pm\infty$	Any value except $\mp\infty$ and NaN	$\pm\infty + x$	$\pm\infty$
	Any value except $\mp\infty$ and NaN	$\pm\infty$	$x + \pm\infty$	
Subtraction	$\pm\infty$	Any value except $\pm\infty$ and NaN	$\pm\infty - x$	
	Any value except $\mp\infty$ and NaN	$\mp\infty$	$x - \mp\infty$	

Although this module may be synthesized with any segmentation value within the range of the natural numbers supported by generic variable VHDL by **LATENCY**, only the first two levels will be exploited. Higher values can be used to match the level of latency with that of other modules but only produced a series of records one after the other within the module which did not improve over the critical path.

2.1.2 fmul

Module responsible for the operation of multiplication of two numbers in single precision floating point.

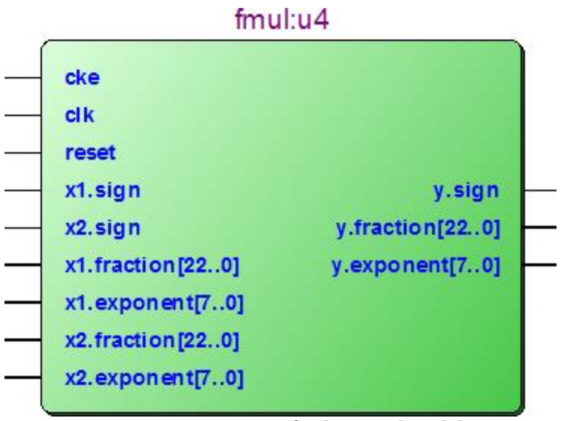


Figure 7. RTL Viewer's fmul external module.

```
entity fmul is
    generic(
        USE_SUBNORMAL        : boolean;
        ROUND_STYLE          : float_round_style;
        LATENCY              : natural;
        EMBEDDED_MULTIPLIER : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        x1 : in pfloat;
        x2 : in pfloat;

        y : out pfloat
    );
end entity;
```

Table 5 shows the operations performed internally by the module and depend on the use or not of the denormalized range are described.

Table 5. Internal Operations module fmul.

Operations	With subnormal range	Without subnormal range
XOR operation with the signs	X	X
Offset correction range dependent denormalized	X	
Sum of exponents with offset correction	X	X
Pre mantissas with normalization carried towards the exponent	X	
Multiplication of the mantissas and standardization led to the result with the exponent	X	X
Rounding-normalized result	X	X
Special cases	X	X

Special cases mentioned in Table 5 are further described in Table 6.

Table 6. Special cases of fmul module.

`x1`	`x2`	Description	Result
NaN	Any value	$\text{NaN} \times x$	NaN
Any value	NaN	$x \times \text{NaN}$	
0	∞	$0 \times \infty$	
∞	0	$\infty \times 0$	
∞	Any value except 0 and NaN	$\infty \times x$	∞
Any value except 0 and NaN	∞	$x \times \infty$	

This particular module has the generic variable **`EMBEDDED_MULTIPLIER`** which lets you select whether the multiplication operation between the fractional parts directly with AND and sums assembled as shown in Figure 8 where numbers are used instead of numbers to 24 bits 4 bits and a 0 latency level for easy viewing, or if instead dedicated multipliers are used (in case the count on said multipliers FPGA) as shown in Figure 9.

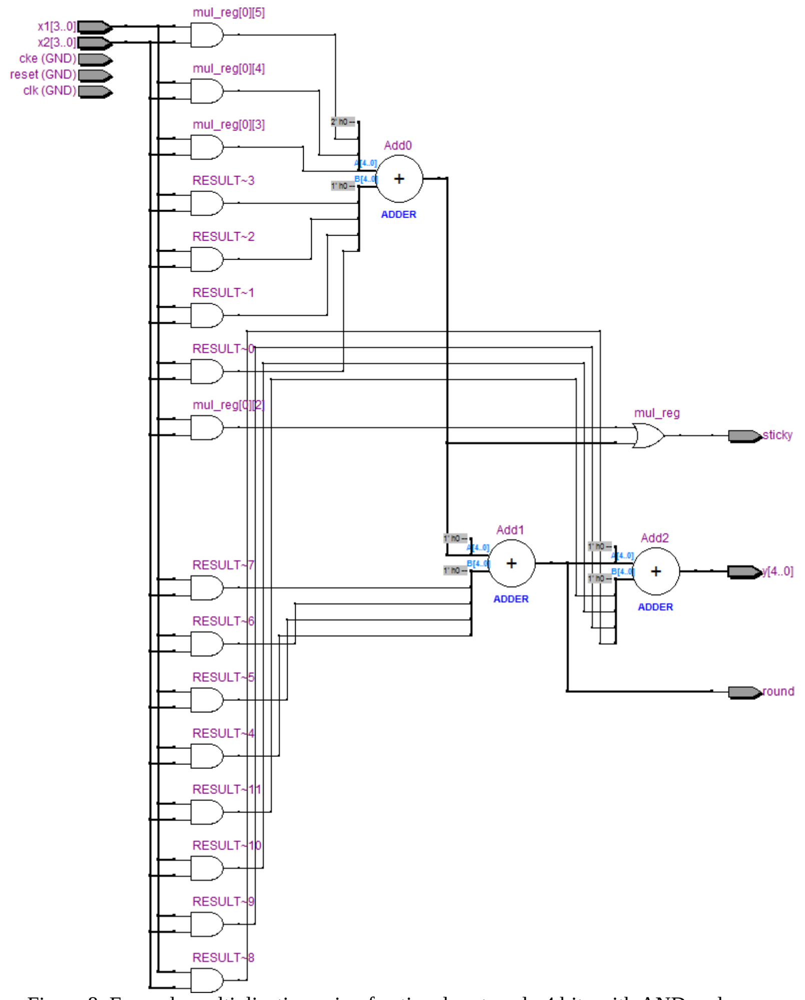


Figure 8. Example multiplication using fractional parts only 4 bits with AND and sums.

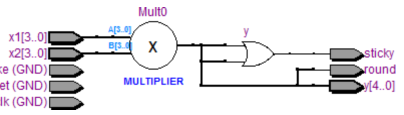


Figure 9. Example multiplication fractional parts using 4-bit embedded multipliers.

The output **`y`** of images Figure 8 and Figure 9 represents the result of the operation with the **`guard`** bit, which is used together with the outputs **`round`** and **`sticky`** for rounding mentioned in Table 5.

Although this module can be synthesized to any value within the range segmentation of natural numbers supported by VHDL generic variable **LATENCY** through the use of levels of segmentation depends on the use or not of embedded multipliers. Failure to use the embedded multipliers can be leveraged up to 24 levels of segmentation, but in case of using only the first 2 levels are utilized. Higher values can be used to match the level of latency with that of other modules but only produced a series of records one after the other within the module which does not improve over the critical path.

2.1.3 fdiv

Module responsible for the operation of division between two numbers in single precision floating point.

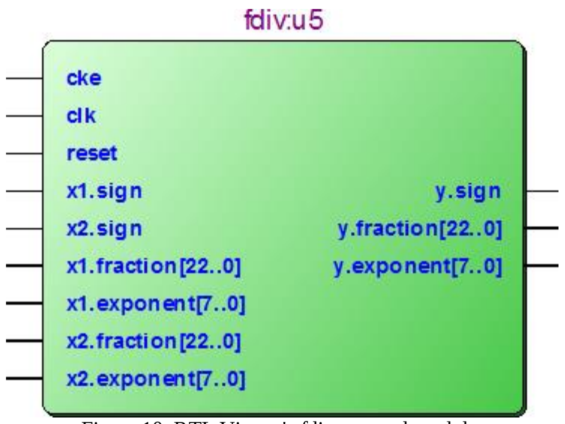


Figure 10. RTL Viewer's fdiv external module.

```
entity fdiv is
    generic(
        USE_SUBNORMAL : boolean;
        ROUND_STYLE   : float_round_style;
        LATENCY       : natural
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        x1 : in pfloat;
        x2 : in pfloat;

        y : out pfloat
    );
end entity;
```

Table 7 lists the operations performed internally by the module and depend on the use or not of the denormalized range are described.

Table 7. Internal Operations module fdiv.

Operations	With subnormal range	Without subnormal range
XOR operation with the signs	X	X
Offset correction range dependent denormalized	X	
Subtracting Exponents with offset correction	X	X
Pre mantissas with normalization carried towards the exponent	X	
Division mantissas and standardization led to the result with the exponent	X	X
Rounding-normalized result	X	X
Special cases	X	X

Special cases listed in Table 7 are further described in Table 8.

Table 8. Special cases of fdiv module.

x1	x2	Description	Result
NaN	Any value	NaN ÷ x	NaN
Any value	NaN	x ÷ NaN	
∞	∞	∞ ÷ ∞	
0	0	0 ÷ 0	
∞	Any value except ∞ and NaN	∞ ÷ x	∞
Any value except 0 and NaN	0	x ÷ 0	

To make the division between the fractional parts of both numbers combinational non-restoration division is used. Internally the module this division is between two 24-bit numbers, but easier to view in Figure 11 shows the same operation between two 4-bit numbers with a level of latency 0, where **y** represents the output result of the operation with the **guard** bit, which is used together with the outputs **round** and **sticky** for rounding mentioned in Table 7.

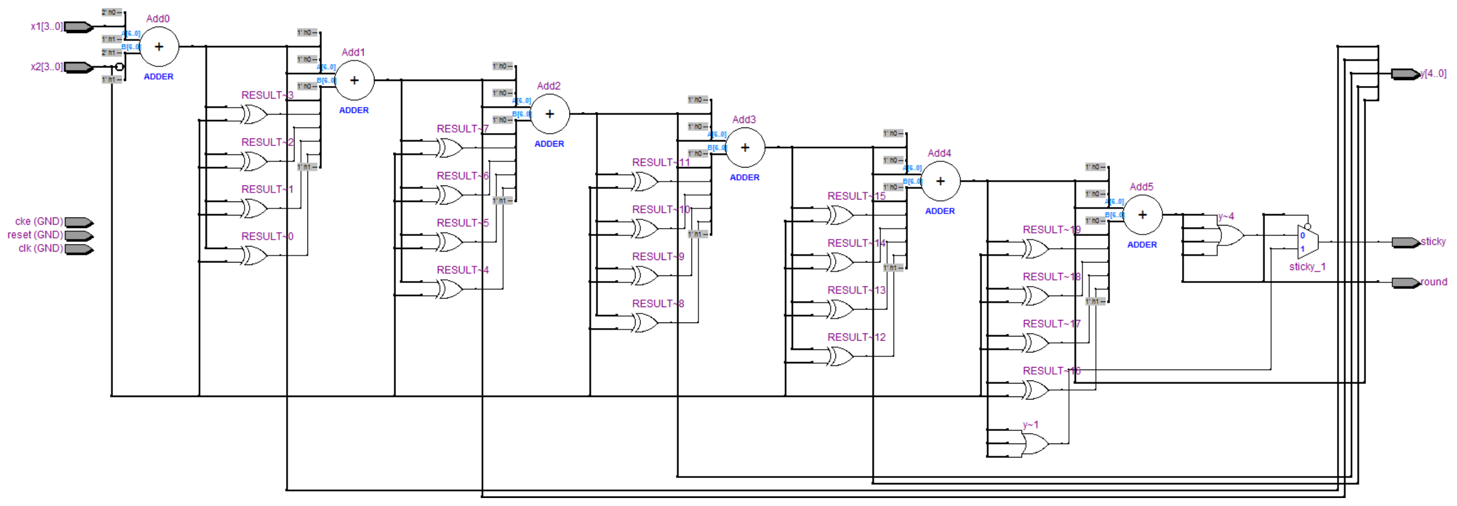


Figure 11. Splitting example using 4-bit fractional parts.

Although this module may be synthesized with any segmentation value within the range of the natural numbers supported by generic variable VHDL **LATENCY** through only the first 27 levels are utilized. Higher values can be used to match the level of latency with that of other modules but only produced a series of records one after the other within the module which does not improve over the critical path.

2.1.4 fmulp2

Module responsible for performing the multiplication operation between a floating-point number of single precision, and a power of two 9-bit signed, that is, two raised to between -256 and 255.

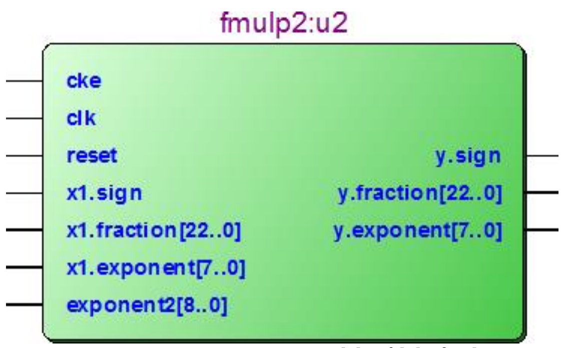


Figure 12. RTL Viewer externo del módulo fmulp2.

```
entity fmulp2 is
    generic(
        USE_SUBNORMAL : boolean;
        LATENCY       : natural
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        x1        : in pfloat;
        exponent2 : in signed(8 downto 0);

        y : out pfloat
    );
end entity;
```

The reason for synthesizing this particular module is that this multiplication does not require segmentation due to its low complexity and therefore times lower latency generated multiplication between two floating-point numbers single precision (fmul), which is can optimize those multiplications involving a power of two used this module.

Furthermore, this module hardware consumes little compared to the other modules.

Table 9. Internal Operations module fmulp2.

Operations	With subnormal range	Without subnormal range
Offset correction range dependent denormalized	X	
Sum of exponents with offset correction	X	X
Mantissa normalization pre carried towards the exponent	X	
Special cases	X	X

2.1.5 fcomp_fmin_fmax

Module in charge of comparing two floating point numbers in single precision and determine if the first (**x1**) is less than, equal to or greater than the second (**x2**) and enables to obtain the minimum of two numbers (fmin) or the maximum of the two numbers (fmax).

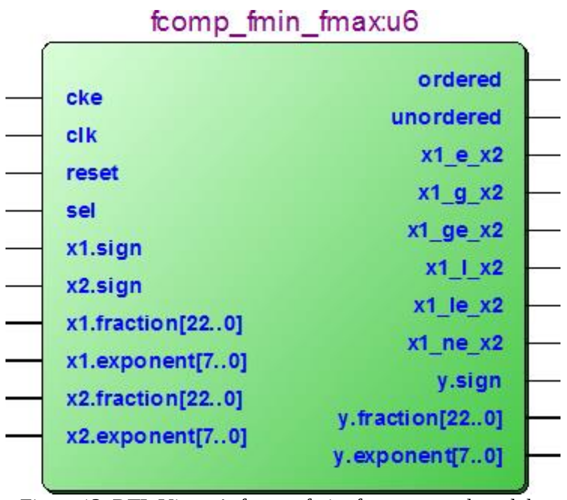


Figure 13. RTL Viewer's fcomp_fmin_fmax external module.

```
entity fcomp_fmin_fmax is
    generic(
        USE_SUBNORMAL : boolean;
        LATENCY_1     : natural;
        LATENCY_2     : natural
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        x1 : in pfloat;
        x2 : in pfloat;

        sel : in std_logic;

        x1_l_x2   : out std_logic;
        x1_le_x2  : out std_logic;
        x1_e_x2   : out std_logic;
        x1_ge_x2  : out std_logic;
        x1_g_x2   : out std_logic;
        x1_ne_x2  : out std_logic;
        ordered   : out std_logic;
        unordered : out std_logic;

        y : out pfloat
    );
end entity;
```

This particular module performs a fairly simple operation, which does not need segmentation, but anyway for purposes of equal levels of latency with other generic modules with 2 latent variables (which do not improve the critical path):

- **`LATENCY_1`** for comparison outputs **`x1_l_x2`**, **`x1_le_x2`**, **`x1_e_x2`**, **`x1_ge_x2`**, **`x1_g_x2`**, **`x1_ne_x2`**, **`ordered`**, **`unordered`**.
- **`LATENCY_2`** to output maximum or minimum **`y`**.

The behavior of the comparator outputs specified in Table 10.

Table 10. Output response comparison of fcomp_fmin_fmax module.

	x1_l_x2	x1_le_x2	x1_e_x2	x1_ge_x2	x1_g_x2	x1_ne_x2	ordered	unordered
x1 < **x2**	'1'	'1'	'0'	'0'	'0'	'1'	'1'	'0'
x1 = **x2**	'0'	'1'	'1'	'1'	'0'	'0'	'1'	'0'
x1 > **x2**	'0'	'0'	'0'	'1'	'1'	'1'	'1'	'0'
only **x1**, only **x2**, or both are NaN	'0'	'0'	'0'	'0'	'0'	'1'	'0'	'1'

And the behavior of the maximum or minimum output specified in Table 11.

Table 11. Behavior of the minimum or maximum output of module fcomp_fmin_fmax.

	sel	Operation	y
x1 < **x2**	'0'	fmin	**x1**
x1 = **x2**			**x2**
x1 > **x2**			**x2**
only **x1** or both are NaN			**x2**
only **x2** is NaN			**x1**
x1 < **x2**	'1'	fmax	**x2**
x1 = **x2**			**x1**
x1 > **x2**			**x1**
only **x1** or both are NaN			**x2**
only **x2** is NaN			**x1**

In case the generic variable **`USE_SUBNORMAL`** false, the values belonging to the range of denormalized numbers are treated as if they were exactly zero, either +0 or -0, regardless of the value of **`USE_SUBNORMAL`** both the value +0 as - 0 are treated as equals: +0 = -0.

2.1.6 trunc_round_ceil_floor

Module responsible for carrying out the operations of truncation (trunc), rounding to the nearest value (round), rounding up (ceil) or rounded down (floor) on a number of single-precision floating point.

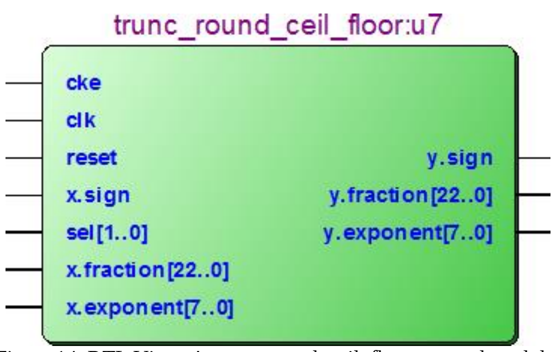


Figure 14. RTL Viewer's trunc_round_ceil_floor external module.

```
entity trunc_round_ceil_floor is
    generic(
        USE_SUBNORMAL : boolean;
        LATENCY       : natural
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        sel : in std_logic_vector(1 downto 0);
        x   : in pfloat;

        y : out pfloat
    );
end entity;
```

The **`sel`** input port determines which of the four operations are carried out, as shown in Table 12.

Table 12. Selection module operation trunc_round_ceil_floor.

sel	Operation
"00"	trunc
"01"	round
"10"	ceil
"11"	floor

If the input value **`x`** is infinite or NaN output will contain the same value as the input regardless of the selected operation.

2.2 core

I module responsible for performing mathematical operations required to perform the graphics processing to be performed. This module contains the modules described above and other modules additions that will be described hereinafter which share **`cke`** input port (to enable or disable the clock signal), **`clk`** (clock) and **`reset`** (returning to its state initial the records contained in the module).

```
entity core is
    generic(
        USE_SUBNORMAL       : boolean;
        ROUND_STYLE         : float_round_style;
        DEDICATED_REGISTERS : boolean;
        LATENCY_1           : boolean;
        FADD_LATENCY        : natural;
        EMBEDDED_MULTIPLIER : boolean;
        FMUL_LATENCY        : natural;
        FDIV_LATENCY        : natural;
        FCOMP_LATENCY       : natural
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        pc          : buffer unsigned(31 downto 0);
        instruction : in     std_logic_vector(31 downto 0);

        address   : out    unsigned(31 downto 0);
        read      : buffer std_logic;
        readdata  : in     std_logic_vector(31 downto 0);
        write     : buffer std_logic;
        writedata : out    std_logic_vector(31 downto 0);

        stop_core : out std_logic;
        irq       : out std_logic
    );
end entity;
```

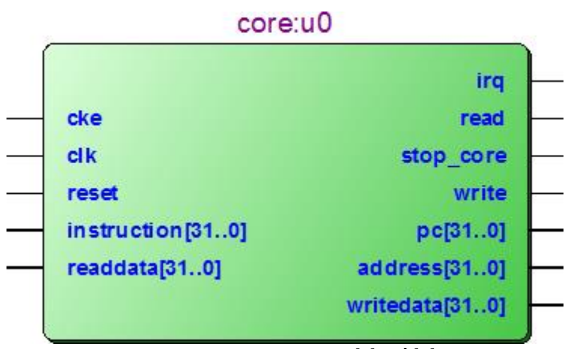


Figure 15. RTL Viewer externo del módulo core.

In core_pkg.vhd file two types of additional data that are used to generate the portc_tristate_bridge module defines:

```
type addrc_t is array(natural range <>) of unsigned(4 downto 0);
type datac_t is array(natural range <>) of std_logic_vector(31 downto 0);
```

A **`to_integer(boolean)`** function is also defined to adapt the generic variable **`LATENCY_1`** those modules that are programmed with a value natural latency and not boolean:

```
function to_integer(x : boolean) return integer is
begin
    if x then
        return 1;
    else
        return 0;
    end if;
end function;
```

The modules contained within the core are generated with the following generic variables:

```
u0 : registers
generic map(
    ADDR_WIDTH=> 5,
    DATA_WIDTH=> 32,
    USE_RESET=>  DEDICATED_REGISTERS
)
                          ⋮
u2 : fmulp2
generic map(
    USE_SUBNORMAL=> USE_SUBNORMAL,
    LATENCY=>       to_integer(LATENCY_1)
)
                          ⋮
u3 : fadd_fsub
generic map(
    USE_SUBNORMAL=> USE_SUBNORMAL,
    ROUND_STYLE=>   ROUND_STYLE,
    LATENCY=>       FADD_LATENCY
)
                          ⋮
u4 : fmul
generic map(
    USE_SUBNORMAL=>       USE_SUBNORMAL,
    ROUND_STYLE=>         ROUND_STYLE,
    LATENCY=>             FMUL_LATENCY,
    EMBEDDED_MULTIPLIER=> EMBEDDED_MULTIPLIER
)
                          ⋮
u5 : fdiv
generic map(
    USE_SUBNORMAL=> USE_SUBNORMAL,
    ROUND_STYLE=>   ROUND_STYLE,
    LATENCY=>       FDIV_LATENCY
)
                          ⋮
u6 : fcomp_fmin_fmax
generic map(
    USE_SUBNORMAL=> USE_SUBNORMAL,
    LATENCY_1=>     FCOMP_LATENCY,
    LATENCY_2=>     to_integer(LATENCY_1)
)
```

```
u7 : trunc_round_ceil_floor
generic map(
    USE_SUBNORMAL=> USE_SUBNORMAL,
    LATENCY=>       to_integer(LATENCY_1)
)
                          ⋮
u8 : ftou8
generic map(
    DATAC_REG=> LATENCY_1
)
                          ⋮
u9 : ftou16
generic map(
    DATAC_REG=> LATENCY_1
)
                          ⋮
u10 : ftou32
generic map(
    DATAC_REG=> LATENCY_1
)
                          ⋮
u11 : u8tof
generic map(
    DATAC_REG=> LATENCY_1
)
                          ⋮
u12 : u16tof
generic map(
    DATAC_REG=> LATENCY_1
)
                          ⋮
u13 : u32tof
generic map(
    DATAC_REG=> LATENCY_1
)
                          ⋮
u14 : portc_tristate_bridge
generic map(
    PORTS=> 17
)
```

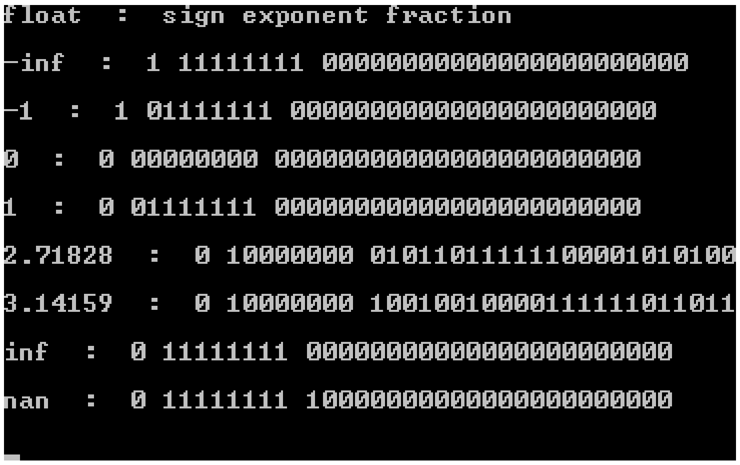
Considering that there are certain values for floating point numbers that could be used much more frequently than the rest, it was decided to include a small ROM that would contain eight relevant floating point values that could quickly be selected instead of loading them from memory program.

These values correspond to negative infinity minus one, zero, one, the Euler number, pi, infinity, and NaN.

But before describing these values as a ROM in vhdl had to know their values in binary format, so they decided to make a small program in C ++:

```
const std::array<float, 8> ks = {
    -std::numeric_limits<float>::infinity(),
    -1.f,
    0.f,
    1.f,
    std::exp(1.f),
    std::acos(-1.f),
    std::numeric_limits<float>::infinity(),
    std::numeric_limits<float>::quiet_NaN(),
};

union float_bits_to_uint32 {
    float_bits_to_uint32(const float &F) : f(F) {}
    operator std::uint32_t() const {return u32;}
private:
    float f;
    std::uint32_t u32;
};
```



```
float   :  sign exponent fraction
-inf   :  1 11111111 00000000000000000000000
-1   :  1 01111111 00000000000000000000000
0   :  0 00000000 00000000000000000000000
1   :  0 01111111 00000000000000000000000
2.71828   :  0 10000000 01011011111100001010100
3.14159   :  0 10000000 10010010000111111011011
inf   :  0 11111111 00000000000000000000000
nan   :  0 11111111 10000000000000000000000
```

Figure 16. Screenshot of the program designed to show the binary format floating point numbers simple presicion.

In the previous code an array containing the 8 values to which they are want to know your binary, and further down the same code a union that is built with a floating point value is observed and can be converted to an integer unsigned 32 bits (such conversion only changes the way the 32-bit without changing its value are interpreted, which is different from the direct conversion type).

Then you just have to walk the array using binding float_bits_to_uint32, the standard template std::bitset and output stream std::cout to display the binary format of the 8 values as shown in Figure 16.

```
std::cout<<"float   :  sign exponent fraction\n\n";

for (i = 0; i < ks.size(); i++) {
    std::cout<<ks[i]<<"  :  "<<std::bitset<32>(float_bits_to_uint32(ks[i])).to_string().insert(9, " ").insert(1, " ")<<"\n\n";
}
```

All instructions of the core may be executed conditionally, depending on the bits 26 to 22 of the instruction and the value contained in 8 **cr** special registers, for which **ie** matrix that determines which instructions are enabled is used in each of the latency levels available as shown in the following code:

```
constant INSTRUCTION_1_LATENCY : natural := maximum(
    maximum(to_integer(LATENCY_1), FADD_LATENCY), maximum(maximum(FMUL_LATENCY, FDIV_LATENCY), FCOMP_LATENCY)
);
                                                    ⋮
type ie_t is array(0 to INSTRUCTION_1_LATENCY) of std_logic_vector(31 downto 0);
                                                    ⋮
signal ie : ie_t;
                                                    ⋮
process(clk, cke, reset,
        instruction,
        instruction_1, cr, ie)
    variable decoder : std_logic_vector(31 downto 0);
    variable ie_1    : std_logic_vector(31 downto 0);
begin
    decoder := (others=> '0');
    decoder(to_integer(unsigned(instruction(31 downto 27)))) := '1';
    ie_1 := (others=> (
        not(instruction(26)) or (instruction(25) xor cr(to_integer(unsigned(instruction(24 downto 22)))))
    ));
    ie(0) <= decoder and ie_1;
    instruction_1(0) <= instruction;

    for i in 1 to INSTRUCTION_1_LATENCY loop
        if reset = '1' then
            ie(i) <= (others=> '0');
            instruction_1(i) <= (others=> '0');
        elsif rising_edge(clk) and (cke = '1') then
            ie(i) <= ie(i-1);
            instruction_1(i) <= instruction_1(i-1);
        end if;
    end loop;
end process;
```

The 26-bit instruction determines if the instruction conditionally or not executed, and if executed conditionally bit 25 determines if the value depends a **cr** or its negated value register and bits 24 to 22 determine which of the 8 **cr** enforcement records or instruction depends.

Besides connecting modules operation logs module through module tri-state bridge, one of the most important tasks of the core is to control access to the memory where program instructions and data memory are.

Access to the data memory is performed by the output ports **address**, **read**, **write** and **wirtedata**, and **readdata** input port as shown in the following code:

```
process(clk, reset, cke,
        pc,
        read, write,
        instruction_1, ie,
        dataa, datab,
        address_1)
    variable address_2 : unsigned(31 downto 0);
begin
    case instruction_1(0)(11) is
    when '0'=>
        address_2 := unsigned(datab);
    when '1'=>
        address_2 := add_sub_f(address_1, to_unsigned(4, 32), instruction_1(0)(10))(31 downto 0);
    when others=>
    end case;

    if reset = '1' then
        address_1 <= (others=> '0');
    elsif rising_edge(clk) and (cke = '1') and ((ie(0)(2#10010#) = '1') or (read = '1') or (write = '1')) then
        address_1 <= address_2;
    end if;

    case instruction_1(0)(12) is
    when '0'=>
        address <= address_2;
    when '1'=>
        address <= address_1;
    when others=>
    end case;

    read <= ie(0)(2#10011#) or ie(0)(2#10111#);
    write <= ie(0)(2#10100#) or ((ie(0)(2#10101#) or ie(0)(2#10110#)) and instruction_1(0)(13));

    if ie(0)(2#10100#) = '1' then
        writedata <= dataa;
    else
        writedata <= std_logic_vector(pc);
    end if;
end process;
```

Bit 12 of the current instruction determines whether the output port **address** corresponds to the special register **address_1** or **address_2** signal is used to update the special register **address_1** a clock cycle after the current 11 bit instruction determines if the signal **address_2** corresponds to the module port records b or an increase or decrease (depending on the bit 10 of the current instruction) **address_1** special register.

This bit can be used 12 to 10 from the current statement to update the special register **address_1** and memory access port to the value b of module records, or to access memory with per-decrement, pre- increases, decreases, or post-post-increments **address_1** special register.

The output ports **read** and **write** are enabled or disabled depending on whether the current instruction is for a read or write data memory and conditional execution depending on the above.

Access to the program memory is done via the **pc** output port and input port **instruction** as shown in the following code:

```
process(clk, cke, reset,
        pc,
        instruction_1,
        ie,
        pc_1)
```

```
    variable k   : std_logic_vector(17 downto 0);
    variable pck : signed(32 downto 0);
begin
    k := instruction_1(0)(21 downto 14)&instruction_1(0)(9 downto 0);
    pck := signed('0'&pc_1)+signed(k);

    if reset = '1' then
        pc <= (others=> '0');
    elsif rising_edge(clk) and (cke = '1') then
        if ie(0)(2#10101#) = '1' then
            pc <= unsigned(pck(31 downto 0));
        elsif ie(0)(2#10110#) = '1' then
            pc <= unsigned(datab);
        elsif ie(0)(2#10111#) = '1' then
            pc <= unsigned(readdata);
        else
            pc <= pc+1;
        end if;
    end if;

    if reset = '1' then
        pc_1 <= (others=> '0');
    elsif rising_edge(clk) and (cke = '1') then
        pc_1 <= pc;
    end if;
end process;
```

For most instructions the **pc** output port corresponding to the program counter is incremented by one unit each clock cycle, but for certain instructions at that counter will be adding a whole lot of 18-bit signed, assign the value b port module registers or the port value **readdata** entry, this also depending on the aforementioned conditional execution.

2.2.1 registers

Container module 32 major registers (each 32 bits) using the core to perform various operations.

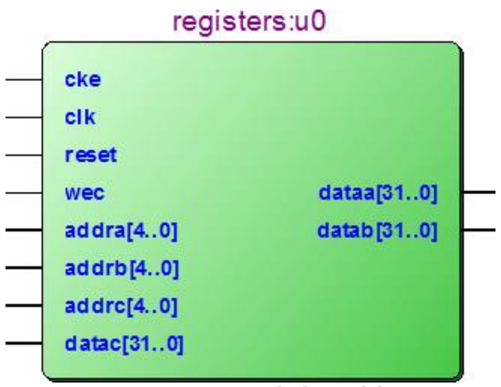


Figure 17. RTL Viewer outside the module registers.

```
entity registers is
    generic(
        ADDR_WIDTH : natural;
        DATA_WIDTH : natural;
        USE_RESET  : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        addra : in  unsigned(ADDR_WIDTH-1 downto 0);
        dataa : out std_logic_vector(DATA_WIDTH-1 downto 0);

        addrb : in  unsigned(ADDR_WIDTH-1 downto 0);
        datab : out std_logic_vector(DATA_WIDTH-1 downto 0);

        addrc : in unsigned(ADDR_WIDTH-1 downto 0);
        wec   : in std_logic;
        datac : in std_logic_vector(DATA_WIDTH-1 downto 0)
    );
end entity;
```

As shown in Figure 17, the module has two read ports and one 32-bit write port also 32 bits.

The main reason for the choice of 32 as the number of records is being sought which could operate a 4x4 matrix which contains 16 numbers with a vector of four components without the storage of records is exceeded in which case there would recourse to the data memory which would make the process very slow, thus the registration module should be able to store a minimum of 20 floating point numbers in single precision, so we decided to have 32 records, with 32 the number power of 2 closest to 20 and higher than the latter.

In case the generic variable is false **USE_RESET** not be possible to reset the records by **reset** input port, which in turn will allow this module is synthesized using records memory modules of the FPGA instead of using dedicated registers.

Memory modules use the FPGA instead of using dedicated registers allows records module can operate at a higher clock frequency.

2.2.2 ftou8

Module in charge of interpreting the 32-bit input port **dataa** as a floating point number in single precision and make it an unsigned integer of 8 bits, combined bits of the result with 24 of the 32 bits of input port **datab**.

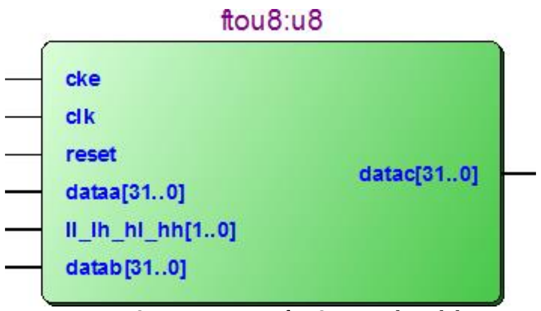


Figure 18. RTL Viewer's ftou8 external module.

```
entity ftou8 is
    generic(
        DATAC_REG : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        dataa       : in std_logic_vector(31 downto 0);
        ll_lh_hl_hh : in std_logic_vector(1 downto 0);
        datab       : in std_logic_vector(31 downto 0);

        datac : out std_logic_vector(31 downto 0)
    );
end entity;
```

To perform the type conversion functions **to_pfloat(std_logic_vector)** and **to_unsigned(pfloat, natural)** described in section pfloat (pipelined float) are used.

```
u8 := to_unsigned(to_pfloat(dataa), 8);
```

The way the 8 bits resulting from the conversion rate and the bits of input port **datab** depends on the input port **ll_lh_hl_hh** combine. The result of this combination corresponds to the 32-bit output port **datac**.

Table 13. Selecting the alignment module ftou8.

ll_lh_hl_hh	Combination
"00"	The 8-bit result of the type conversion correspond to bits 7 to 0 of the **datac** output port, other bits correspond to the respective bits of the **datab** input port: `datac <= datab(31 downto 8)&std_logic_vector(u8);`
"01"	The 8-bit result of the type conversion correspond to bits 15 to 8 of the **datac** output port, other bits correspond to the respective bits of the **datab** input port: `datac <= datab(31 downto 16)&std_logic_vector(u8)&datab(7 downto 0);`
"10"	The 8-bit result of the type conversion correspond to bits 23 to 16 of the **datac** output port, other bits correspond to the respective bits of the **datab** input port: `datac <= datab(31 downto 24)&std_logic_vector(u8)&datab(15 downto 0);`
"11"	The 8-bit result of the type conversion correspond to bits 31 to 24 of the **datac** output port, other bits correspond to the respective bits of the **datab** input port: `datac <= std_logic_vector(u8)&datab(23 downto 0);`

If the **DATAC_REG** generic variable is true the signals **clk**, **reset** and **cke** are used for create a register in the **datac** output port.

2.2.3 ftou16

Module in charge of interpreting the 32-bit input port **dataa** as a floating point number in single precision and make it an unsigned integer of 16 bits combined bits of the result with 16 of the 32 bits of input port **datab**.

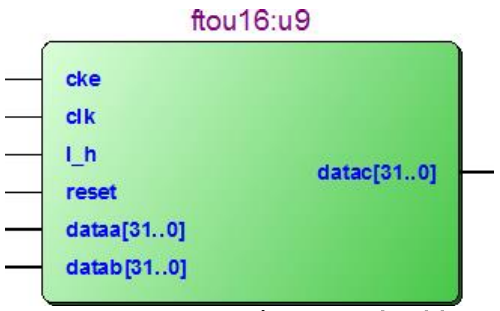


Figure 19. RTL Viewer's ftou16 external module.

```
entity ftou16 is
    generic(
        DATAC_REG : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        dataa : in std_logic_vector(31 downto 0);
        l_h   : in std_logic;
        datab : in std_logic_vector(31 downto 0);

        datac : out std_logic_vector(31 downto 0)
    );
```

```
end entity;
```

To perform the type conversion functions **`to_pfloat(std_logic_vector)`** and **`to_unsigned(pfloat, natural)`** described in section pfloat (pipelined float) are used.

```
u16 := to_unsigned(to_pfloat(dataa), 16);
```

The way you combine the 16-bit result of the conversion rate and the bits of input port **`datab`** **`l_h`** depends on the input port. The result of this combination corresponds to the 32-bit output port **`datac`**.

Table 14. Selecting the alignment module ftou16.

`l_h`	**Combination**
`'0'`	The 16-bit result of the type conversion correspond to bits 15 to 0 of the **`datac`** output port, other bits correspond to the respective bits of the **`datab`** input port: `datac <= datab(31 downto 16)&std_logic_vector(u16);`
`'1'`	The 16-bit result of the type conversion correspond to bits 31 to 16 of the **`datac`** output port, other bits correspond to the respective bits of the **`datab`** input port: `datac <= std_logic_vector(u16)&datab(15 downto 0);`

If the **`DATAC_REG`** generic variable is true the signals **`clk`**, **`reset`** and **`cke`** are used for create a register in the **`datac`** output port.

2.2.4 ftou32

Module responsible for interpreting the 32-bit input port **`dataa`** as a floating point number in single precision and make it an unsigned integer of 32 bits. The result of the conversion rate applicable to the 32-bit output port **`datac`**.

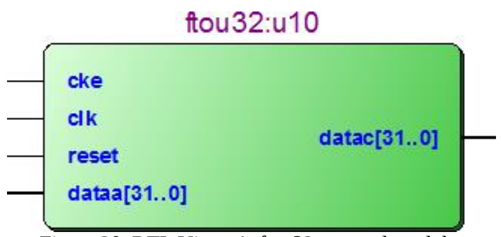


Figure 20. RTL Viewer's ftou32 external module.

```
entity ftou32 is
    generic(
        DATAC_REG : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        dataa : in  std_logic_vector(31 downto 0);
        datac : out std_logic_vector(31 downto 0)
    );
end entity;
```

To perform the type conversion functions **`to_pfloat(std_logic_vector)`** and **`to_unsigned(pfloat, natural)`** described in section pfloat (pipelined float) are used.

```
datac <= std_logic_vector(to_unsigned(to_pfloat(dataa), 32));
```

If the **`DATAC_REG`** generic variable is true the signals **`clk`**, **`reset`** and **`cke`** are used for create a register in the **`datac`** output port.

2.2.5 u8tof

Module that interprets 8 of the 32 bits of input port **`datab`** as an unsigned integer into a floating point number in single precision. The result of the conversion rate applicable to the 32-bit output port **`datac`**.

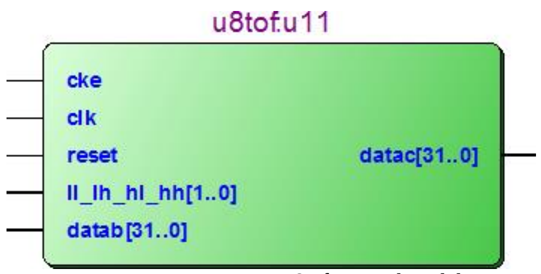


Figure 21. RTL Viewer's u8tof external module.

```
entity u8tof is
    generic(
        DATAC_REG : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        ll_lh_hl_hh : in std_logic_vector(1 downto 0);
        datab       : in std_logic_vector(31 downto 0);

        datac : out std_logic_vector(31 downto 0)
    );
end entity;
```

The way the bits corresponding to the selected integer unsigned 8-bit from 32-bit input port **datab** **ll_lh_hl_hh** depends on the input port.

Table 15. Selecting the alignment module u8tof.

ll_lh_hl_hh	Selection
"00"	Bits 7 to 0 of the **datab** input port correspond to unsigned integer of 8-bit: `u8 := unsigned(datab(7 downto 0));`
"01"	Bits 15 to 8 of the **datab** input port correspond to unsigned integer of 8-bit: `u8 := unsigned(datab(15 downto 8));`
"10"	Bits 23 to 16 of the **datab** input port correspond to unsigned integer of 8-bit: `u8 := unsigned(datab(23 downto 16));`
"11"	Bits 31 to 24 of the **datab** input port correspond to unsigned integer of 8-bit: `u8 := unsigned(datab(31 downto 24));`

To perform the type conversion functions **to_pfloat(unsigned, natural)** and **to_stdlogicvector(pfloat)** described in pfloat (pipelined float) section is used.

```
datac <= to_stdlogicvector(to_pfloat(u8));
```

If the **DATAC_REG** generic variable is true the signals **clk**, **reset** and **cke** are used for create a register in the **datac** output port.

2.2.6 u16tof

Module that interprets 16 of the 32 bits of input port **datab** as an unsigned integer into a floating point number in single precision. The result of the conversion rate applicable to the 32-bit output port **datac**.

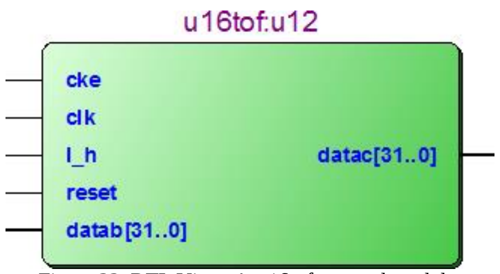


Figure 22. RTL Viewer's u16tof external module.

```
entity u16tof is
    generic(
        DATAC_REG : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        l_h   : in std_logic;
        datab : in std_logic_vector(31 downto 0);

        datac : out std_logic_vector(31 downto 0)
    );
end entity;
```

The way the bits corresponding to the selected integer unsigned 16-bit from 32-bit input port **datab** depends **l_h** input port.

Table 16. Selecting the alignment module u16tof.

l_h	Selection
'0'	Bits 15 to 0 of the **datab** input port correspond to unsigned integer of 16 bits: `u16 := unsigned(datab(15 downto 0));`
'1'	Bits 31 to 16 of the **datab** input port correspond to unsigned integer of 16 bits: `u16 := unsigned(datab(31 downto 16));`

To perform the type conversion functions **to_pfloat(unsigned, natural)** and **to_stdlogicvector(pfloat)** described in pfloat (pipelined float) section is used.

```
datac <= to_stdlogicvector(to_pfloat(u16, 2));
```

If the **DATAC_REG** generic variable is true the signals **clk**, **reset** and **cke** are used for create a register in the **datac** output port.

2.2.7 u32tof

Module responsible for interpreting the 32-bit input port **datab** as an unsigned integer into a floating point number in single precision. The result of the conversion rate applicable to the 32-bit output port **datac**.

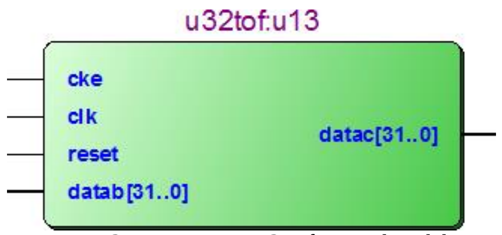


Figure 23. RTL Viewer's u32tof external module.

```
entity u32tof is
    generic(
        DATAC_REG : boolean
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;
        cke   : in std_logic;

        datab : in  std_logic_vector(31 downto 0);
        datac : out std_logic_vector(31 downto 0)
    );
end entity;
```

To perform the type conversion functions **to_pfloat(unsigned, natural)** and **to_stdlogicvector(pfloat)** described in pfloat (pipelined float) section is used.

```
datac <= to_stdlogicvector(to_pfloat(unsigned(datab), 4));
```

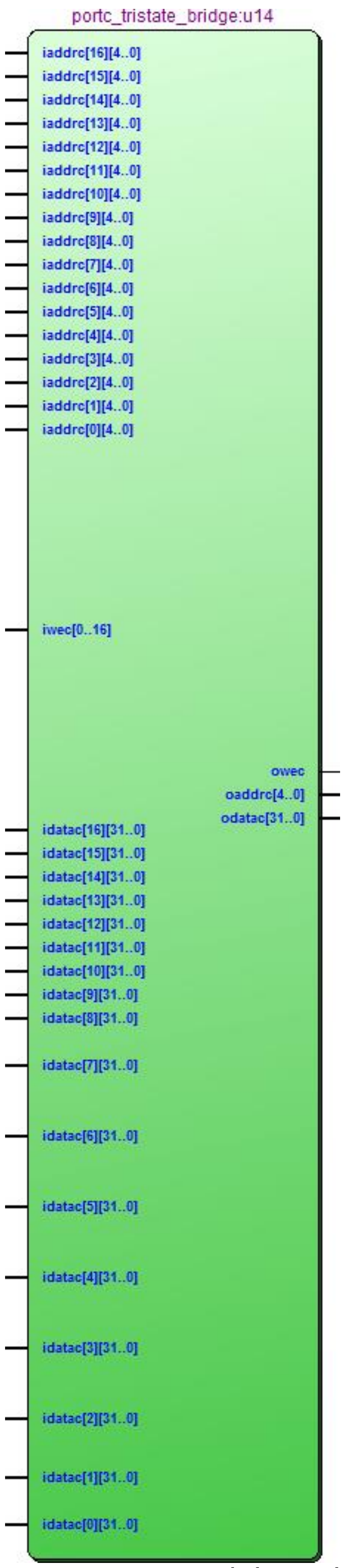
If the **DATAC_REG** generic variable is true the signals **clk**, **reset** and **cke** are used for create a register in the **datac** output port.

2.2.8 portc_tristate_bridge

I module responsible for connecting the outputs of the different modules of operation of the core to the write port module records the core.

We decided to use a tri-state bridge instead of a multiplexer to allow operation modules may have different levels of segmentation, which in turn produces different levels of latency in the result yielded by these modules.

During the design of the graphics processing unit initially sought that all modules have the same level of latency, from which is derived the reason that many modules have the option to adjust the latency even if this already no further improvement in the critical path is obtained, but at some point the design was decided to give the possibility of different levels of latency.



```
entity portc_tristate_bridge is
    generic(
        PORTS : natural
    );
    port(
        iaddrc : in addrc_t(0 to PORTS-1);
        iwec   : in std_logic_vector(0 to PORTS-1);
        idatac : in datac_t(0 to PORTS-1);

        oaddrc : out unsigned(4 downto 0);
        owec   : out std_logic;
        odatac : out std_logic_vector(31 downto 0)
    );
end entity;
```

Figure 24. RTL Viewer's portc_tristate_bridge external module.

The consequence of having modules operating at different levels of latency is that several modules of operation may result in trying to write a module registers simultaneously, so instead of creating a module registers with multiple ports writing or designing a other hardware control strategy, it was decided to delegate the responsibility to prevent this from happening in the software.

In other words, the instructions stored in the memory of the gpu module program must be organized in such a way that at no time more modules operating simultaneously attempt to write the result in the module registers.

However, the major advantage of allowing different levels of latency segmentation is used only those who really need modules.

2.3 gpu

Module container program memory and graphics processing core, which in turn serves as an interface to the Avalon bus and can be easily integrated to the Nios II processor using the Qsys tool.

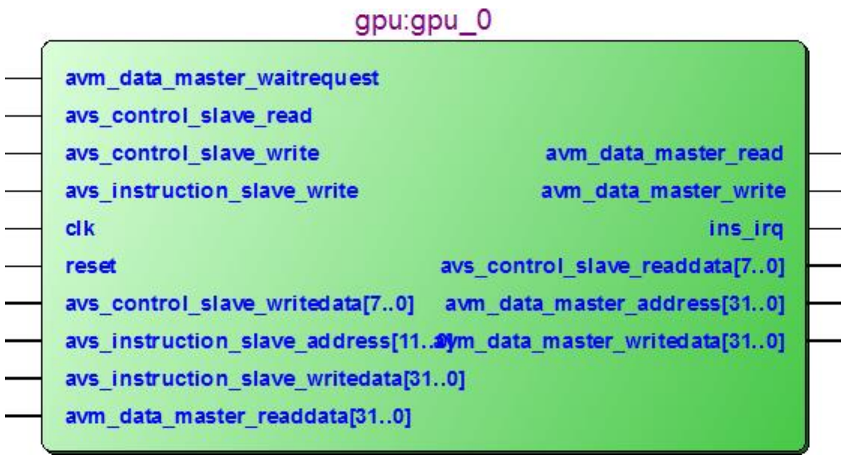

gpu:gpu_0
avm_data_master_waitrequest
avs_control_slave_read
avs_control_slave_write
avs_instruction_slave_write
clk
reset
avs_control_slave_writedata[7..0]
avs_instruction_slave_address[11..0]
avs_instruction_slave_writedata[31..0]
avm_data_master_readdata[31..0]
avm_data_master_read
avm_data_master_write
ins_irq
avs_control_slave_readdata[7..0]
avm_data_master_address[31..0]
avm_data_master_writedata[31..0]

Figure 25. External gpu RTL Viewer module.

```
entity gpu is
    generic(
        INSTRUCTION_SLAVE_ADDRESS_WIDTH : natural range 0 to 32
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;

        avs_control_slave_read      : in  std_logic;
        avs_control_slave_readdata  : out std_logic_vector(7 downto 0);
        avs_control_slave_write     : in  std_logic;
        avs_control_slave_writedata : in  std_logic_vector(7 downto 0);

        avs_instruction_slave_address   : in std_logic_vector(INSTRUCTION_SLAVE_ADDRESS_WIDTH-1 downto 0);
        avs_instruction_slave_write     : in std_logic;
        avs_instruction_slave_writedata : in std_logic_vector(31 downto 0);

        avm_data_master_address     : out    std_logic_vector(31 downto 0);
        avm_data_master_read        : buffer std_logic;
        avm_data_master_readdata    : in     std_logic_vector(31 downto 0);
        avm_data_master_write       : buffer std_logic;
        avm_data_master_writedata   : out    std_logic_vector(31 downto 0);
        avm_data_master_waitrequest : in     std_logic;

        ins_irq : out std_logic
    );
end entity;
```

The instance of the core contained in this module is generated with generic variables shown in the following code:

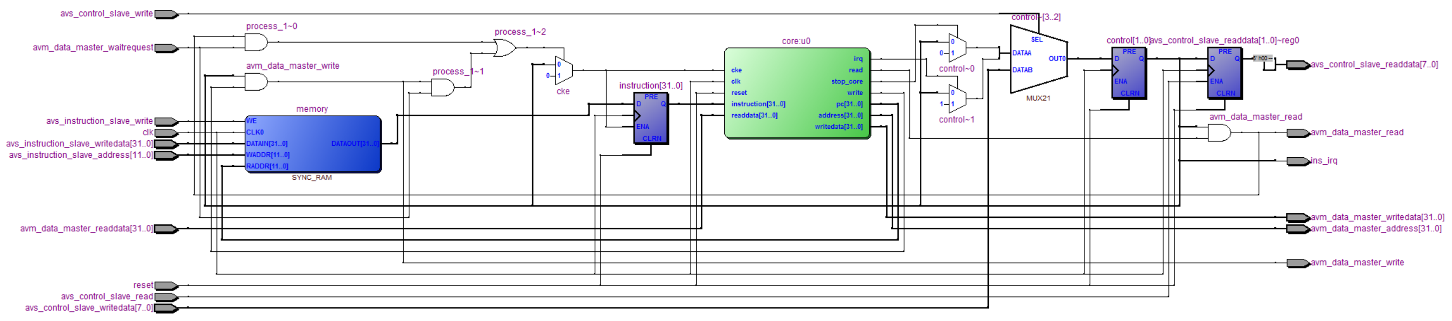
```
u0 : core
generic map(
    USE_SUBNORMAL=>        false,
    ROUND_STYLE=>          round_to_nearest,
    DEDICATED_REGISTERS=> false,
    LATENCY_1=>            false,
    FADD_LATENCY=>         2,
    EMBEDDED_MULTIPLIER=> true,
    FMUL_LATENCY=>          2,
    FDIV_LATENCY=>          10,
    FCOMP_LATENCY=>         0
)
```

In Table 17 is shown in detail using the gpu hardware module and each of the modules containing (some generated automatically when certain operation is performed) using the FPGA on the card EP2C70F896C6 development DE2-70.

Table 17. Hardware using.

Entity	Logic Cells	Dedicated Logic Registers	Memory Bits	M4Ks	DSP Elements	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
gpu:gpu_0	4331 (128)	1018 (116)	133270	35	7	3283 (8)	234 (60)	814 (53)
altsyncram:memory_rtl_0	0 (0)	0 (0)	131072	32	0	0 (0)	0 (0)	0 (0)
altsyncram_rgi1:auto_generated	0 (0)	0 (0)	131072	32	0	0 (0)	0 (0)	0 (0)
core:u0	4211 (457)	902 (132)	2198	3	7	3275 (302)	174 (28)	762 (129)
altshift_taps:instruction_1_rtl_0	16 (0)	7 (0)	150	1	0	9 (0)	0 (0)	7 (0)
shift_taps_e1n:auto_generated	16 (1)	7 (1)	150	1	0	9 (0)	0 (0)	7 (1)
altsyncram_mta1:altsyncram2	0 (0)	0 (0)	150	1	0	0 (0)	0 (0)	0 (0)
cntr_okf:cntr1	8 (8)	3 (3)	0	0	0	5 (5)	0 (0)	3 (3)
cmpr_7cc:cmpr8								
cntr_e4h:cntr3	7 (7)	3 (3)	0	0	0	4 (4)	0 (0)	3 (3)
registers:u0	146 (146)	80 (80)	2048	2	0	61 (61)	38 (38)	47 (47)
altsyncram:regs_rtl_0	0 (0)	0 (0)	1024	1	0	0 (0)	0 (0)	0 (0)
altsyncram_37i1:auto_generated	0 (0)	0 (0)	1024	1	0	0 (0)	0 (0)	0 (0)
altsyncram:regs_rtl_1	0 (0)	0 (0)	1024	1	0	0 (0)	0 (0)	0 (0)
altsyncram_37i1:auto_generated	0 (0)	0 (0)	1024	1	0	0 (0)	0 (0)	0 (0)
fmulp2:u2	31 (31)	0 (0)	0	0	0	28 (28)	0 (0)	3 (3)
fadd_fsub:u3	624 (404)	75 (47)	0	0	0	549 (357)	1 (1)	74 (46)
add_sub:u0	220 (220)	28 (28)	0	0	0	192 (192)	0 (0)	28 (28)
fmul:u4	194 (132)	69 (42)	0	0	7	125 (90)	0 (0)	69 (42)
mul:u0	62 (7)	27 (27)	0	0	7	35 (6)	0 (0)	27 (1)
lpm_mult:Mult0	55 (0)	0 (0)	0	0	7	29 (0)	0 (0)	26 (0)
mult_p8t:auto_generated	55 (55)	0 (0)	0	0	7	29 (29)	0 (0)	26 (26)
fdiv:u5	1527 (116)	539 (45)	0	0	0	985 (66)	107 (2)	435 (59)
div:u0	1413 (1413)	494 (494)	0	0	0	919 (919)	105 (105)	389 (389)
fcomp_fmin_fmax:u6	69 (69)	0 (0)	0	0	0	69 (69)	0 (0)	0 (0)
trunc_round_ceil_floor:u7	115 (115)	0 (0)	0	0	0	114 (114)	0 (0)	1 (1)
ftou8:u8	44 (44)	0 (0)	0	0	0	44 (44)	0 (0)	0 (0)
ftou16:u9	60 (60)	0 (0)	0	0	0	60 (60)	0 (0)	0 (0)
ftou32:u10	79 (79)	0 (0)	0	0	0	79 (79)	0 (0)	0 (0)
u8tof:u11	28 (28)	0 (0)	0	0	0	28 (28)	0 (0)	0 (0)
u16tof:u12	76 (76)	0 (0)	0	0	0	75 (75)	0 (0)	1 (1)
u32tof:u13	142 (142)	0 (0)	0	0	0	141 (141)	0 (0)	1 (1)
portc_tristate_bridge:u14	649 (649)	0 (0)	0	0	0	606 (606)	0 (0)	43 (43)

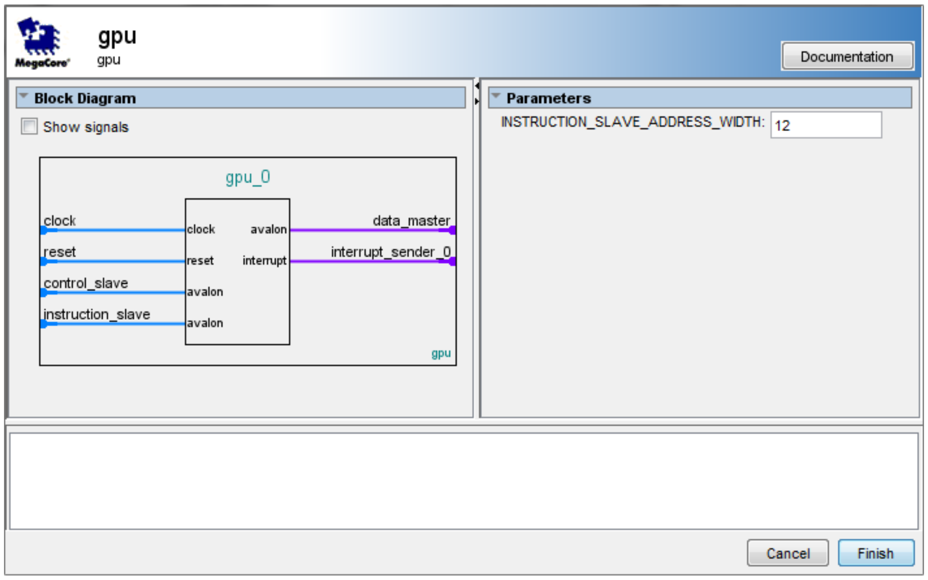
In Figure 26 an overview of the gpu module is shown using the RTL Viewer tool Quartus II.



avs_control_slave_write
avm_data_master_waitrequest
process_1~0
process_1~2
avm_data_master_write
process_1~1
cke
instruction[31..0]
core:u0
control~0
control~1
MUX21
control[1..0]
avs_control_slave_readdata[1..0]~reg0
avs_control_slave_readdata[7..0]
avm_data_master_read
ins_irq
memory
SYNC_RAM
avs_instruction_slave_write
clk
avs_instruction_slave_writedata[31..0]
avs_instruction_slave_address[11..0]
avm_data_master_readdata[31..0]
avm_data_master_writedata[31..0]
avm_data_master_address[31..0]
reset
avs_control_slave_read
avs_control_slave_writedata[7..0]

Figure 26. Internal gpu RLT Viewer module.

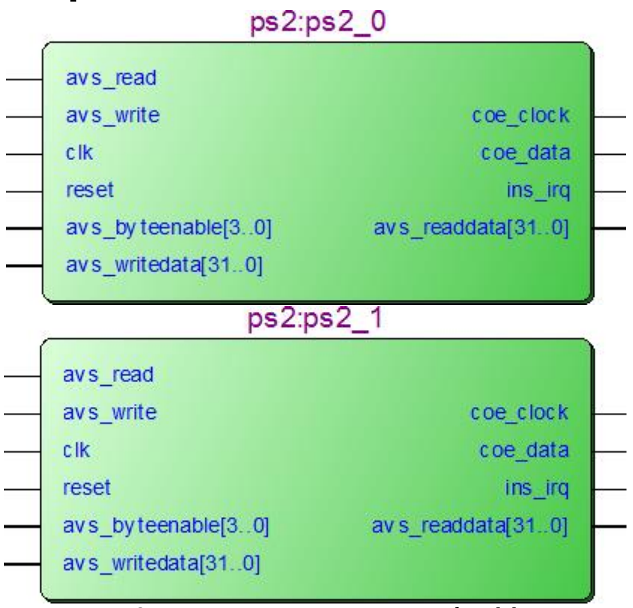
In Figure 27 the dialog box that appears when you want to integrate the module using the Qsys tool Quartus II is shown.



INSTRUCTION_SLAVE_ADDRESS_WIDTH: 12

Figure 27. gpu dialog module in Qsys.

2.4 ps2



```
entity ps2 is
    generic(
        clk_frequency : natural
    );
    port(
        clk   : in std_logic;
        reset : in std_logic;

        avs_byteenable : in  std_logic_vector(3 downto 0);
        avs_read       : in  std_logic;
        avs_readdata   : out std_logic_vector(31 downto 0);
        avs_write      : in  std_logic;
        avs_writedata  : in  std_logic_vector(31 downto 0);

        ins_irq : out std_logic;

        coe_clock : inout std_logic;
        coe_data  : inout std_logic
    );
end entity;
```

Figure 28. RTL Viewer outer two instances of module ps2.

Because the version of Quartus II 11.1sp2 a module for communication with the keyboard and mouse are used during work was not available through the ps/2 protocol we decided to design and implement a module for this purpose.

This module is instantiated 2 times where coe_clock outputs and coe_data instance ps2_0 are connected to pins of the FPGA that correspond to PS2_KBDAT and PS2_KBCLK signals on the development board, and coe_clock outputs and coe_data instance ps2_1 is connect to pins of the FPGA that correspond to signals PS2_MSCLK PS2_MSDAT and development card.

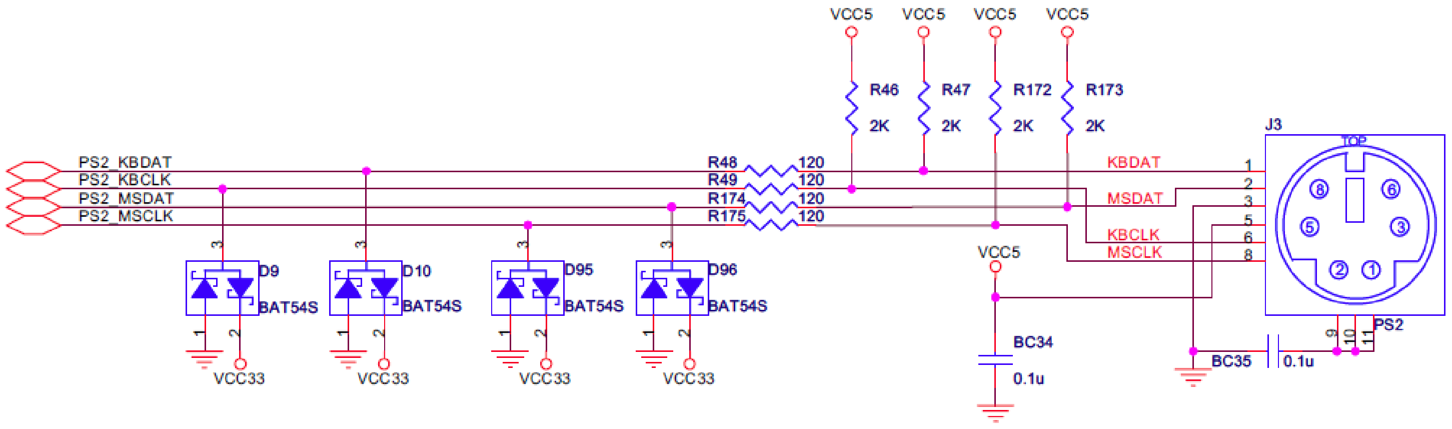


Figure 29. PS/2 schematic (image taken from the manual of the card DE2-70).

Table 18. Map pin PS/2 external module.

Signal	Pin	Signal	Pin
PS2_KBCLK	PIN_F24	PS2_MSCLK	PIN_D26
PS2_KBDAT	PIN_E24	PS2_MSDAT	PIN_D25

3 Compiler assembler code for the graphics processing unit

Due to the complexity of the graphics processing to be performed, it is necessary to have a set of instructions that control the behavior of the hardware modules for each of the 4 stages of the process, but at the same time, because the whole set of instructions required is quite large (around 2500) it is virtually impossible to express directly in binary form, so there is the need to design and implement a compiler assembly code that translates instructions from a friendlier format with the programmer (assembler) to the binary format that directly receives the graphics processor.

The core module of the graphics processing unit is initially turned off waiting for the instructions needed to operate on the program memory, which once loaded does not require you to be modified at any time to load.

Then proceed to charge data memory arrangement of vertices to be processed along with the transformation matrix to be applied, among other values.

Finally through the Avalon bus can give instruction to initiate the graphics processing which is done entirely by the gpu module by executing instructions stored in the program memory of the module without additional external actions are required.

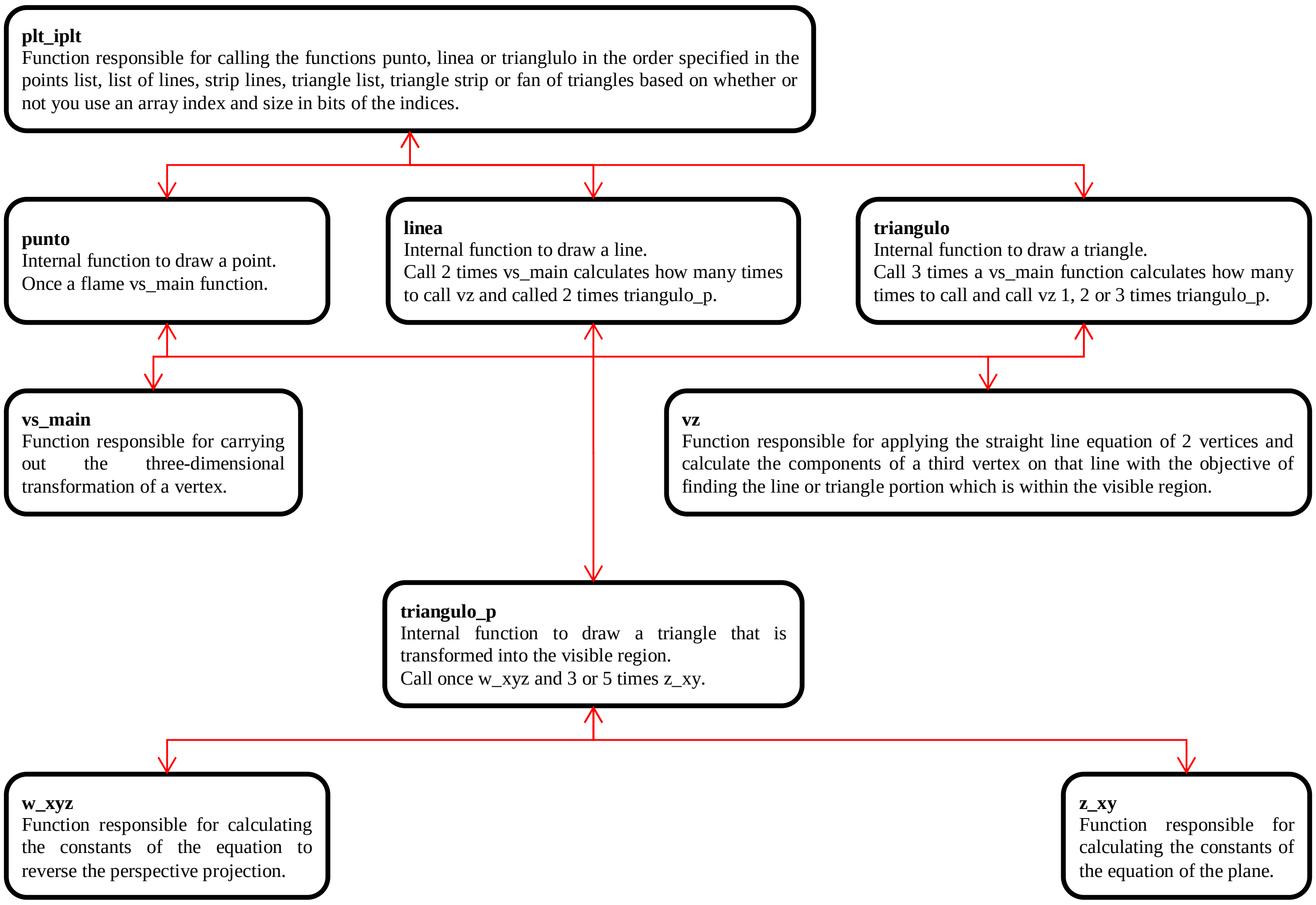

plt_iplt
Function responsible for calling the functions punto, linea or trianglulo in the order specified in the points list, list of lines, strip lines, triangle list, triangle strip or fan of triangles based on whether or not you use an array index and size in bits of the indices.
punto
Internal function to draw a point.
Once a flame vs_main function.
linea
Internal function to draw a line.
Call 2 times vs_main calculates how many times to call vz and called 2 times triangulo_p.
triangulo
Internal function to draw a triangle.
Call 3 times a vs_main function calculates how many times to call and call vz 1, 2 or 3 times triangulo_p.
vs_main
Function responsible for carrying out the three-dimensional transformation of a vertex.
vz
Function responsible for applying the straight line equation of 2 vertices and calculate the components of a third vertex on that line with the objective of finding the line or triangle portion which is within the visible region.
triangulo_p
Internal function to draw a triangle that is transformed into the visible region.
Call once w_xyz and 3 or 5 times z_xy.
w_xyz
Function responsible for calculating the constants of the equation to reverse the perspective projection.
z_xy
Function responsible for calculating the constants of the equation of the plane.

Figure 30. Block diagram of the overall structure of the program in assembler graphics processing unit.

GPU Assembler

Instruction	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
nop	00000					condition																										
ploadf_l loadf_h ploadu_l loadu_h ploadaddr_l loadaddr_h	00001					condition					k																l_h	rc				
copy fabs fneg fnabs	00010					condition					k_ra	ra		k							sel							rc				
fmulp2	00011					condition					k_ra	ra		k					p2									rc				
fadd fsub	00100					condition					k_ra	ra		k								sel	rb					rc				
fmul	00101					condition					k_ra	ra		k									rb					rc				
fdiv	00110					condition					k_ra	ra		k									rb					rc				
fmin fmax	00111					condition					k_ra	ra		k								sel	rb					rc				
trunc round ceil floor	01000					condition					k_ra	ra		k							sel							rc				
ftou8_ll ftou8_lh ftou8_hl ftou8_hh	01001					condition					k_ra	ra		k							sel		rb					rc				
ftou16_l ftou16_h	01010					condition					k_ra	ra		k								sel	rb					rc				
ftou32	01011					condition					k_ra	ra		k														rc				
u8tof_ll u8tof_lh u8tof_hl u8tof_hh	01100					condition															sel		rb					rc				
u16tof_l u16tof_h	01101					condition																sel	rb					rc				
u32tof	01110					condition																	rb					rc				
fcomp_l fcomp_le fcomp_e fcomp_ge fcomp_g fcomp_ne fcomp_o fcomp_u	01111					condition					k_ra	ra		k						sel			rb							cr		
add	10000					condition					1	ra											rb					rc				
load_addr	10001					condition																						rc				
store_addr	10010					condition														000			rb									
d_load i_load, pop load load_d load_i	10011					condition														sel			rb					rc				
d_store i_store store store_d, push store_i	10100					condition					k_ra	ra		k						sel			rb									
jump rcall	10101					condition					k								sel	111			k									
goto call	10110					condition													sel	111			rb									
ret	10111					condition														010												
stop_core	11000					condition																										
irq	11001					condition																										

Figure 31. Format assembly instructions for graphics processing unit.

At the time in which he finished writing in the memory section data corresponding to the two-dimensional space are projected mentioned vertices in principle, an interrupt is sent through the Avalon bus and switches off the core module.

Notably, the act of writing the resulting image in a section of the data memory does not mean that this is displayed on the screen, this task falls to other hardware modules external to the graphics processing unit.

3.1 nop

This instruction does nothing and there in order to wait for further instructions meet their latency so they can make their operating properly.

3.2 ploadu_l, ploadf_l, ploadaddr_l, loadu_h, loadf_h, loadaddr_h

Click responsible first to preload the least significant 16 bits of an integer constant, floating point, or a memory address (ploadu_l, ploadf_l and ploadaddr_l respectively) in a special register also 16 bits, later combine it with the 16-bit most significant of an integer constant, floating point, or a memory address (loadu_h, loadf_h and loadaddr_h respectively) and send the result to the module containing the 32 normal records.

Table 19. Example of ploadu_l, ploadf_l, ploadaddr_l, loadu_h, loadf_h, loadaddr_h instructions.

	Instruction	Operation performed
Integer constant	`ploadu_l 10;` `loadu_h 10, r0;`	`uint32_t r0 = 10;`
Constant in single precision floating point	`ploadf_l 3.141592654;` `loadf_h 3.141592654, r0;`	`float r0 = 3.141592654;`
Memory address	`ploadaddr_l 1024;` `loadaddr_h 1024, r0;`	`uint32_t r0 = 1024;`

The only difference between using the full version and the version constant memory address of the instruction is that if the memory addresses compiler to mark the instruction corresponding to the memory address bits they add value corresponding to the base address of the data memory of the gpu.

3.3 copy, fabs, fneg, fnabs

Click responsible for copying the data from one record to another, copy the absolute value of the data, multiplied by -1 to copy the data, and copy -1 multiplied by the absolute value of the data respectively.

Table 20. Example of copy, fabs, fneg, fnabs instrucions.

Instruction	Operation performed
`copy r0, r1;`	`r1 = r0;`
`fabs r0, r1;`	`r1 = abs(r0);`
`fneg r0, r1;`	`r1 = -r0;`
`fnabs r0, r1;`	`r1 = -abs(r0);`

3.4 fmulp2

Instruction responsible for multiplying a floating-point number by an integer power of 2 in the range from -256 to 255 and store the result in a register.

Table 21. Example of fmulp2 instruction.

Instruction	Operation performed
`fmulp2 r0, -4, r1;`	`r1 = r0*pow(2, -4);`
`fmulp2 r0, 0, r1;`	`r1 = r0*pow(2, 0);`
`fmulp2 r0, 8, r1;`	`r1 = r0*pow(2, 8);`

3.5 fadd, fsub

Click responsible respectively add and subtract two numbers in floating point and store the result in a register after they have passed two clock cycles.

Table 22. Example of fadd, fsub instructions.

Instruction	Operation performed
`fadd r0, r1, r2;`	`r2 = r0+r1;`
`nop;`	
`nop;`	
`fsub r0, r1, r2;`	`r2 = r0-r1;`
`fadd r0, r1, r0;`	`r0 = r0+r1;`
`nop;`	
`nop;`	
`fsub r0, r1, r0;`	`r0 = r0-r1;`
`fadd r0, r1, r1;`	`r1 = r0+r1;`
`nop;`	
`nop;`	
`fsub r0, r1, r1;`	`r1 = r0-r1;`
`fadd r0, r0, r1;`	`r1 = r0+r0;`
`nop;`	
`nop;`	
`fsub r0, r0, r1;`	`r1 = r0-r0;`
`fadd r0, r0, r0;`	`r0 = r0+r0;`
`nop;`	
`nop;`	
`fsub r0, r0, r0;`	`r0 = r0-r0;`

3.6 fmul

Instruction responsible for multiplying two floating point numbers and store the result in a register after they have passed two clock cycles.

Table 23. Example of fmul instruction.

Instruction	Operation performed
`fmul r0, r1, r2;`	`r2 = r0*r1;`
`fmul r0, r1, r0;`	`r0 = r0*r1;`
`fmul r0, r1, r1;`	`r1 = r0*r1;`
`fmul r0, r0, r1;`	`r1 = r0*r0;`
`fmul r0, r0, r0;`	`r0 = r0*r0;`

3.7 fdiv

Instruction in charge of dividing two floating point numbers and store the result in a register after they have spent 10 clock cycles.

Table 24. Example of fdiv instruction.

Instruction	Operation performed
`fdiv r0, r1, r2;`	`r2 = r0/r1;`
`fdiv r0, r1, r0;`	`r0 = r0/r1;`
`fdiv r0, r1, r1;`	`r1 = r0/r1;`
`fdiv r0, r0, r1;`	`r1 = r0/r0;`
`fdiv r0, r0, r0;`	`r0 = r0/r0;`

3.8 fmax, fmin

Click responsible for calculating maximum and minimum respectively between two floating point numbers, and store the result in a register.

Table 25. Example of fmax, fmin instructions.

Instruction	Operation performed
`fmax r0, r1, r2;`	`r2 = fmax(r0, r1);`
`fmin r0, r1, r2;`	`r2 = fmin(r0, r1);`
`fmax r0, r1, r0;`	`r0 = fmax(r0, r1);`
`fmin r0, r1, r0;`	`r0 = fmin (r0, r1);`
`fmax r0, r1, r1;`	`r1 = fmax(r0, r1);`
`fmin r0, r1, r1;`	`r1 = fmin (r0, r1);`
`fmax r0, r0, r1;`	`r1 = fmax(r0, r0);`
`fmin r0, r0, r1;`	`r1 = fmin (r0, r0);`
`fmax r0, r0, r0;`	`r0 = fmax(r0, r0);`
`fmin r0, r0, r0;`	`r0 = fmin (r0, r0);`

3.9 trunc, round, ceil, floor

Click responsible for truncating, rounding to the nearest integer value, rounding to the higher integer and rounding to the lower integer value, respectively, a floating point number and store the result in a register.

Table 26. Example of trunc, round, ceil, floor instructions.

Instruction	Operation performed
`trunc r0, r1;`	`r1 = trunc(r0);`
`round r0, r1;`	`r1 = round(r0);`
`ceil r0, r1;`	`r1 = ceil(r0);`
`floor r0, r1;`	`r1 = floor(r0);`
`trunc r0, r0;`	`r0 = trunc(r0);`
`round r0, r0;`	`r0 = round(r0);`
`ceil r0, r0;`	`r0 = ceil(r0);`
`floor r0, r0;`	`r0 = floor(r0);`

3.10 ftou8_ll, ftou8_lh, ftou8_hl, ftou8_hh

Click responsible for converting a floating point number to an integer unsigned 8 bits and store the result in 8 of the 32 bits of a register. The ftou8_ll instruction stores the result in bits 0 to 7, ftou8_lh in bits 8 to 15, ftou8_hl in bits 16 to 23 and ftou8_hh in bits 24 to 31.

Table 27. Example of ftou8_ll, ftou8_lh, ftou8_hl, ftou8_hh instructions.

Instruction	Operation performed
	`float r0; struct _s {uint8_t ll, lh, hl, hh;} r1;`
`ftou8_ll r0, r1, r1;`	`r1.ll = uint8_t(r0);`
`ftou8_lh r0, r1, r1;`	`r1.lh = uint8_t(r0);`
`ftou8_hl r0, r1, r1;`	`r1.hl = uint8_t(r0);`
`ftou8_hh r0, r1, r1;`	`r1.hh = uint8_t(r0);`
	`union _u {float f; struct _s {uint8_t ll, lh, hl, hh;} s;} r0;`
`ftou8_ll r0, r0, r0;`	`r0.s.ll = uint8_t(r0.f);`
`ftou8_lh r0, r0, r0;`	`r0.s.lh = uint8_t(r0.f);`
`ftou8_hl r0, r0, r0;`	`r0.s.hl = uint8_t(r0.f);`
`ftou8_hh r0, r0, r0;`	`r0.s.hh = uint8_t(r0.f);`

3.11 ftou16_l, ftou16_h

Click responsible for converting a floating point number to an integer unsigned 16 bits and store the result in 16 of the 32 bits of a register. The ftou16_l instruction stores the result in bits 0 through 15, ftou16_h in bits 16 to 31.

Table 28. Example of ftou16_l, ftou16_h instructions.

Instruction	Operation performed
	`float r0; struct _s {uint16_t l, h;} r1;`
`ftou16_l r0, r1, r1;`	`r1.l = uint16_t(r0);`
`ftou16_h r0, r1, r1;`	`r1.h = uint16_t(r0);`
	`union _u {float f; struct _s {uint16_t l, h;} s;} r0;`
`ftou16_l r0, r0, r0;`	`r0.s.l = uint16_t(r0.f);`
`ftou16_h r0, r0, r0;`	`r0.s.h = uint16_t(r0.f);`

3.12 ftou32

Instruction in charge of converting a floating point number to an integer unsigned 32-bit and store the result in a register.

Table 29. Example of ftou32 instruction.

Instruction	Operation performed
	`float r0; uint32_t r1;`
`ftou32 r0, r1;`	`r1 = uint32_t(r0);`
	`union _u {float f; uint32_t u32;} r0;`
`ftou32 r0, r0;`	`r0.u32 = uint16_t(r0.f);`

3.13 u8tof_ll, u8tof_lh, u8tof_hl, u8tof_hh

Click responsible for converting an unsigned integer of 8 bits in a floating point number and store the result in a register. The u8tof_ll statement processes the bits 0 through 7 check bits u8tof_lh 8 to 15, u8tof_hl bits 16 to 23 and u8tof_hh bits 24 to 31.

Table 30. Example of u8tof_ll, u8tof_lh, u8tof_hl, u8tof_hh instructions.

Instruction	Operation performed
	`struct _s {uint8_t ll, lh, hl, hh;} r0; float r1;`
`u8tof_ll r0, r1;`	`r1 = float(r0.ll);`
`u8tof_lh r0, r1;`	`r1 = float(r0.lh);`
`u8tof_hl r0, r1;`	`r1 = float(r0.hl);`
`u8tof_hh r0, r1;`	`r1 = float(r0.hh);`
	`union _u {struct _s {uint8_t ll, lh, hl, hh;} s; float f;} r0;`
`u8tof_ll r0, r0;`	`r0.f = float (r0.s.ll);`
`u8tof_lh r0, r0;`	`r0.f = float (r0.s.lh);`
`u8tof_hl r0, r0;`	`r0.f = float (r0.s.hl);`
`u8tof_hh r0, r0;`	`r0.f = float (r0.s.hh);`

3.14 u16tof_l, u16tof_h

Click responsible for converting an unsigned integer of 16 bits in a floating point number and store the result in a register. The u16tof_l statement processes the bits 0 to 15 of check bits u16tof_h 16 to 31.

Table 31. Example of u16tof_l, u16tof_h instructions.

Instruction	Operation performed
	`struct _s {uint16_t l, h;} r0; float r1;`
`u16tof_l r0, r1;`	`r1 = float(r0.l);`
`u16tof_h r0, r1;`	`r1 = float(r0.h);`
	`union _u {struct _s {uint16_t l, h;} s; float f;} r0;`
`u16tof_l r0, r0;`	`r0.f = float (r0.s.l);`
`u16tof_h r0, r0;`	`r0.f = float (r0.s.h);`

3.15 u32tof

Instruction in charge of converting an unsigned integer of 32 bits in a floating point number and store the result in a register.

Table 32. Example of u32tof instruction.

Instruction	Operation performed
	`uint32_t r0; float r1;`
`u32tof r0, r1;`	`r1 = float(r0);`
	`union _u {uint32_t u32; float f;} r0;`
`u32tof r0, r0;`	`r0.f = float (r0.u32);`

3.16 fcomp_l, fcomp_le, fcomp_e, fcomp_ge, fcomp_g, fcomp_ne, fcomp_o, fcomp_u

Click responsible for comparing two floating point numbers and store the (true or false) Boolean result in one of the 8 records for comparison. The fcomp_l instruction determines if the first number is less than the second, fcomp_le if less or equal fcomp_e if equal, fcomp_ge if greater or equal fcomp_g if greater, fcomp_ne if different, fcomp_o determines whether any of the two values fcomp_u corresponds to NaN and determines if any of the two values corresponding to NaN.

Table 33. Example of fcomp_l, fcomp_le, fcomp_e, fcomp_ge, fcomp_g, fcomp_ne, fcomp_o, fcomp_u instructions.

Instruction	Operation performed
	`float r0, r1; bool cr0;`
`fcomp_l r0, r1, cr0;`	`cr0 = (r0 < r1);`
`fcomp_le r0, r1, cr0;`	`cr0 = (r0 <= r1);`
`fcomp_e r0, r1, cr0;`	`cr0 = (r0 == r1);`
`fcomp_ge r0, r1, cr0;`	`cr0 = (r0 >= r1);`
`fcomp_g r0, r1, cr0;`	`cr0 = (r0 > r1);`
`fcomp_ne r0, r1, cr0;`	`cr0 = (r0 != r1);`
`fcomp_o r0, r1, cr0;`	`cr0 = (!isnan(r0) && !isnan(r1));`
`fcomp_u r0, r1, cr0;`	`cr0 = (isnan(r0) || isnan(r1));`
	`float r0; bool cr0;`
`fcomp_l r0, r0, cr0;`	`cr0 = (r0 < r0);`
`fcomp_le r0, r0, cr0;`	`cr0 = (r0 <= r0);`
`fcomp_e r0, r0, cr0;`	`cr0 = (r0 == r0);`
`fcomp_ge r0, r0, cr0;`	`cr0 = (r0 >= r0);`
`fcomp_g r0, r0, cr0;`	`cr0 = (r0 > r0);`
`fcomp_ne r0, r0, cr0;`	`cr0 = (r0 != r0);`
`fcomp_o r0, r0, cr0;`	`cr0 = (!isnan(r0) && !isnan(r0));`
`fcomp_u r0, r0, cr0;`	`cr0 = (isnan(r0) || isnan(r0));`

3.17 add

Instruction responsible for adding two integers and store the result in a register.

Table 34. Example of add instruction.

Instruction	Operation performed
`add r0, r1, r2;`	`r2 = r0+r1;`
`add r0, r1, r0;`	`r0 = r0+r1;`
`add r0, r1, r1;`	`r1 = r0+r1;`
`add r0, r0, r1;`	`r1 = r0+r0;`
`add r0, r0, r0;`	`r0 = r0+r0;`

3.18 load_addr

Responsible for reading the instruction pointer to the data memory and store it in a register.

Table 35. Example of load_addr instruction.

Instruction	Operation performed
`load_addr r0;`	`r0 = p;`

3.19 store_addr

Instruction in charge of writing the pointer to the data memory, the data stored in a register.

Table 36. Exmaple of store_addr instruction.

Instruction	Operation performed
`store_addr r0;`	`p = r0;`

3.20 d_load, i_load, pop, load, load_d, load_i

Click responsible to read a data memory and store data in a register. The load instruction first writes an entry in the pointer to the data memory and then reads the data to which the pointer points to and stores it in a register. The d_load instruction reads data by performing a pre-decrement the pointer to the data memory, performing a pre-i_load increase load_d conducting a post-decrement and load_i conducting a post-increment. The pop instruction is equivalent to the i_load instruction and is used to extract an element from the stack.

Table 37. Example of d_load, i_load, pop, load, load_d, load_i instructions.

Instruction	Operation performed
load r0, r1;	r1 = *(p = r0);
load r0, r0;	r0 = *(p = r0);
d_load r0;	r0 = *(--p);
i_load r0;	r0 = *(++p);
pop r0;	r0 = *(++p);
load_d r0;	r0 = *(p--);
load_i r0;	r0 = *(p++);

3.21 d_store, i_store, store, store_d, push, store_i

Click responsible for writing a data in the data memory. The first store instruction writes a pointer to data in the data memory and then writes an entry in the address to which the pointer points. The d_store instruction writes the data by performing a pre-decrement the pointer to the data memory, performing a pre-i_store increase store_d conducting a post-decrement and store_i conducting a post-increment. The push instruction is equivalent to store_d instruction and is used to insert an item in the stack.

Table 38. Example of d_store, i_store, store, store_d, push, store_i instructions.

Instruction	Operation performed
store r0, r1;	*(p = r1) = r0;
store r0, r0;	*(p = r0) = r0;
d_store r0;	*(--p) = r0;
i_store r0;	*(++p) = r0;
store_d r0;	*(p--) = r0;
push r0;	*(p--) = r0;
store_i r0;	*(p++) = r0;

3.22 jump, rcall

Click responsible score a signed integer value to 18-bit program counter. The jump instruction takes no additional action, the statement rcall besides modifying the program counter also writes the address to which the pointer points to the data memory corresponding to the program counter value plus one.

The jump instruction is used to generate relative jumps in the flow of the program which allows you to run or large chunks of code sections or repeat this as many times as necessary. The rcall instruction is also used to generate relative jumps in the flow of the program but also allows you to resume the original stream using the ret instruction.

The jump is done after you have passed one clock cycle, which means that you get executed the next instruction before the jump, so the nop instruction can be used after the jump to avoid errors in the implementation of code.

Table 39. Example of jump, rcall instructions.

Instruction	Operation performed
if_1:	if (cr0) {
[!cr0] jump else_1;	
[!cr0] nop;	
//if code	//if code
jump end_if_1;	
nop;	
else_1:	} else {
//else code	//else code
end_if_1:	}
while_1:	while (cr0) {
//while code	//while code
[cr0] jump while_1;	}
[cr0] nop;	
	int main() {
rcall function_1;	//function();
nop;	}
function_1:	void function() {
//function code	//function code
ret;	}
nop;	

3.23 goto, call

Click responsible for writing a data into the program counter. The goto instruction takes no additional action, the call instruction in addition to modifying the program counter also writes the address to which the pointer points to the data memory corresponding to the program counter value plus one.

They work very similarly to the jump, rcall instructions, only instead of receiving as input a label for a given relative jump record as an argument for an absolute jump. Originally included to perform jumps whose distance was too great for them to jump, rcall instructions, but in the end it was not necessary to use them.

3.24 ret

Instruction in charge of reading the data to which the pointer points to the data memory and stored in the program counter. It is used to return the original flow of the program that has been altered by the rcall instruction or call instruction.

3.25 stop_core

Instruction responsible for stopping the operation of the core. It is used to complete the task of graphics processing to wait until the core is activated again when a new task is ready to be executed.

3.26 irq

Instruction responsible for generating an interrupt. It is used to signal through Avalon bus graphics processing task is completed.

3.27 Example of the use of assembly code for the graphics processing

Then the function used in the code in Figure 30 vs_main responsible for processing three-dimensional displays, which receives through the register r0 the direction of the vertex to be transformed, through the register r1 and the address where you want to store the transformed vertex:

Table 40. Function code used in the Figure 30 vs_main.

Instruction	Operation performed
`vs_main:`	`void vs_main() {`
`#def raddr r0`	
`#def waddr r1`	
`#def x r2`	
`#def y r3`	
`#def z r4`	
`#def w r5`	
`#def _11 r6`	
`#def _12 r7`	
`#def _13 r8`	
`#def _14 r9`	
`#def _21 r10`	
`#def _22 r11`	
`#def _23 r12`	
`#def _24 r13`	
`#def _31 r14`	
`#def _32 r15`	
`#def _33 r16`	
`#def _34 r17`	
`#def _41 r18`	
`#def _42 r19`	
`#def _43 r20`	
`#def _44 r21`	
`ploadaddr_l _11_addr;`	
`loadaddr_h _11_addr, _11;`	
`load _11, _11;`	`_11 = *(  p = _11_addr);`
`i_load _12;`	`_12 = *(++p          );`
`i_load _13;`	`_13 = *(++p          );`
`i_load _14;`	`_14 = *(++p          );`
`i_load _21;`	`_21 = *(++p          );`
`i_load _22;`	`_22 = *(++p          );`
`i_load _23;`	`_23 = *(++p          );`
`i_load _24;`	`_24 = *(++p          );`
`i_load _31;`	`_31 = *(++p          );`
`i_load _32;`	`_32 = *(++p          );`

```
    i_load _33;                          _33 = *(++p          );
    i_load _34;                          _34 = *(++p          );
    i_load _41;                          _41 = *(++p          );
    i_load _42;                          _42 = *(++p          );
    i_load _43;                          _43 = *(++p          );
    i_load _44;                          _44 = *(++p          );

    load raddr, x;                       x = *(  p = raddr);
    i_load y;                            y = *(++p        );
    i_load z;                            z = *(++p        );

    fmul x, _11, _11;                    _11 = x*_11;
    fmul x, _12, _12;                    _12 = x*_12;
    fmul x, _13, _13;                    _13 = x*_13;
    fmul x, _14, _14;                    _14 = x*_14;
    fmul y, _21, _21;                    _21 = y*_21;
    fmul y, _22, _22;                    _22 = y*_22;
    fmul y, _23, _23;                    _23 = y*_23;
    fmul y, _24, _24;                    _24 = y*_24;
    fadd _11, _21, _11;                  _11 = _11+_21;
    fadd _12, _22, _12;                  _12 = _12+_22;
    fadd _13, _23, _13;                  _13 = _13+_23;
    fadd _14, _24, _14;                  _14 = _14+_24;
    fmul z, _31, _31;                    _31 = z*_31;
    fmul z, _32, _32;                    _32 = z*_32;
    fmul z, _33, _33;                    _33 = z*_33;
    fmul z, _34, _34;                    _34 = z*_34;
    fadd _11, _31, _11;                  _11 = _11+_31;
    fadd _12, _32, _12;                  _12 = _12+_32;
    fadd _13, _33, _13;                  _13 = _13+_33;
    fadd _14, _34, _14;                  _14 = _14+_34;
    fadd _11, _41, x;                    x = _11+_41;
    fadd _12, _42, y;                    y = _12+_42;
    fadd _13, _43, z;                    z = _13+_43;
    fadd _14, _44, w;                    w = _14+_44;
    nop;
    nop;

#undef _11
#undef _12
#undef _13
#undef _14
#undef _21
#undef _22
#undef _23
#undef _24
#undef _31
#undef _32
#undef _33
#undef _34
#undef _41
#undef _42
#undef _43
#undef _44
#def color r6
#def blue r7
#def green r8
#def red r9
#def alpha r10
                                         if (cr6) {
    [cr6] i_load blue;                       blue  = *(++p);
    [cr6] i_load green;                      green = *(++p);
                                         } else {
    [!cr6] i_load color;                     color = *(++p);
    [!cr6] u8tof_ll color, blue;             blue  = float(color.ll);
    [!cr6] u8tof_lh color, green;            green = float(color.lh);
    [!cr6] u8tof_hl color, red;              red   = float(color.hl);
    [!cr6] u8tof_hh color, alpha;            alpha = float(color.hh);
                                         }

                                         if (cr0) {
    [cr0] store x, waddr;                    *(  p = waddr) = x;
    [cr0] i_store y;                         *(++p        ) = y;
    [cr0] i_store z;                         *(++p        ) = z;
    [cr0] i_store w;                         *(++p        ) = w;
    [cr0] i_store blue;                      *(++p        ) = blue;
    [cr0] i_store green;                     *(++p        ) = green;
    [cr0] i_store red;                       *(++p        ) = red;
    [cr0] i_store alpha;                     *(++p        ) = alpha;
```

```
#undef raddr
#undef waddr
#undef x
#undef y
#undef z
#undef w
#undef color
#undef blue
#undef green
#undef red
#undef alpha

store_addr stack;
ret;
load_addr stack;
```

```
    }













}
```

Notably, the value of the constant _11_addr has been previously defined and represents the direction of the first 4x4 matrix element used to perform the three-dimensional transformations.

Cr6 special register determines whether the vertex to be transformed vertex corresponds to a spatial and color vector or a vector space vertex and texture coordinates, and the special register cr0 determines whether to write the vertex transformed into the address pointed to by register r1, since for example the function point of Figure 30 can use this result directly from the r2, r3, r4, r5, r7, r8, r9 and r10 records.

Because the pointer to the data memory is modified within the vs_main function must be stored in a defined record previously as stack the address of the stack obtained by calling this function by rcall instruction, to restore its value by store_addr function before using the ret instruction, after which it will store the new value of the stack pointer in the stack register.

As explained above in Section portc_tristate_bridge assembly code must be organized in such a way that at no time more instructions try to write while in the module records in this vein, the instructions which you have to pay more attention are those with higher latency times to 0, in this case fmul fadd and instructions; as seen in the code, the instructions are one after the other because they have the same latency and end of this block of instruction two nop instructions are placed to not interfere with the following instructions have latency 0 .

Another thing to consider is the data dependency where there sure before using a register where the result is stored are met the latency of the instruction will store the result.

This example shows more clearly the combination between using specialized hardware and software running on the hardware that is performed to carry out the graphics processing.

4 Driver Software Graphics Processing Unit

4.1 gpu_vector.h, gpu_vector.cpp

Records managers define 3 types of row vectors:

- **gpu_vector2:** structure representing a two-component vector floating point (x, y).
- **gpu_vector3:** structure representing a three component vector floating point (x, y, z).
- **gpu_vector4:** structure representing a four-component vector floating point (x, y, z, w).

For the 3 types of vectors are also defined an empty constructor and one that receives the respective amount of floating-point arguments, the unary operators + and -, multiplication by a scalar on the right and left, the division by a scalar, the addition, subtraction, multiplication and division with another vector of the same size, the length member function that determines the magnitude of the vector, the comparison operators alike and different (for vectors of the same size) and the operation of scalar product (dot).

For three component vector gpu_vector3 additionally also cross product operation (cross) is defined.

4.2 gpu_matrix.h, gpu_vector.cpp

Files in charge of defining the 4x4 transformation matrix, which has an empty constructor and one with 16 floating point arguments that make up the matrix, the unitary operators + and -, multiplication by a scalar on the right and left, the division by a scalar, addition, subtraction and multiplication of matrices and multiplication by a row vector of four components.

Additionally static member functions are defined to generate the identity matrix, the matrix scaling, translation, rotation axis "x" in "y" and the "z" axis and perspective projection.

4.3 gpu_program.h

File where the program is defined in binary format that will be sent to the program memory of the graphics processing unit.

This file is generated by the compiler assembly code.

4.4 gpu.h, gpu.cpp

Main controller software files that define the structure to write the correct values in program memory and data graphics processing unit and the control register of said unit.

Initially the structure constructor takes as arguments pointers to four functions that define how to read and write through cache, the address of the control register and program memory and data graphics processing unit, the width and height of the drawing area and directions to the color buffer and depth.

Once constructed the structure other member functions can be used:

- **world_view_projection:** Functions responsible for reading and writing the transformation matrix of the data memory.
- **texture:** Internal function to associate a texture to the drawing process the gpu.
- **grosor:** Function responsible for writing data into memory the width of points and lines to draw.
- **caras_sh:** Functions responsible for reading and writing the value that determines whether the triangles with vertices will be drawn clockwise.
- **caras_sah:** Functions responsible for reading and writing the value that determines whether the draw triangles with vertices counterclockwise.
- **ancho:** Functions responsible for reading and writing the width of drawing area.
- **alto:** Functions responsible for reading and writing the height of drawing area.
- **use_z:** Functions responsible for reading and writing the value that determines whether or not to use the depth buffer in the drawing process.

- **use_alpha:** Functions responsible for reading and writing the value that determines whether or not to use transparency information contained in the vertices or texture associated.
- **buffer_addr:** Functions responsible for reading and writing the address of the color buffer.
- **bufferZ_addr:** Functions responsible for reading and writing the address of the depth buffer.
- **borrar:** Function responsible for deleting a certain value the color buffer and/or the depth buffer.
- **plt:** Functions responsible for initiating the process drawn from a set of vertices with position and color, or with position and texture coordinates.
- **iplt:** Functions responsible for initiating the process drawn from a set of indices and vertices with position and color, or with position and texture coordinates.
- **reset_irq:** Internal function to reset the log generated by the interruption.

4.4.1 General description of the task startup graphics processing

As mentioned above, the first step is to initialize the software driver to call the constructor of the gpu structure, which among other parameters are passed pointers to four functions that indicate how to pass through the cache memory of the Nios II if I did, this in order to ensure that when given the order to the initial gpu graphics processing all required data has been written into the data memory and have not been in the cache the Nios II which does not have access the gpu.

These functions must be defined by the programmer and not part of the driver files software, this in order that the software does not rely on other than the standard C ++ functions.

Analyzing documentation Nios II processor can reach the following code for the 4 required functions:

```
uint8_t  bypass_cache_read8 (volatile const void *addr) {return IORD_8DIRECT (addr, 0);}
uint32_t bypass_cache_read32(volatile const void *addr) {return IORD_32DIRECT(addr, 0);}
void bypass_cache_write8 (volatile void *addr, uint8_t  data) {IOWR_8DIRECT (addr, 0, data);}
void bypass_cache_write32(volatile void *addr, uint32_t data) {IOWR_32DIRECT(addr, 0, data);}
```

After having defined these functions and built the gpu structure member functions described may be used initially to define the values of the 4x4 transformation matrix, delete the target memory section, among other things.

Having defined all these properties simply represent values in the data memory can begin the task of graphics processing with the plt functions for vertex arrays or vertex iplt for arrays and indices.

Below is the code template used private member function is shown to generate plt public member functions, which is similar to the private member function template iplt:

```
template<bool use_texture, typename T>
void _plt(const PLT &tipo, const T *const&vs_data, const uint32_t &vs_size,
          const uint32_t &inicio, const uint32_t &count) {
    if ((sizeof(vs_data)*CHAR_BIT > 32) &&
        (uintptr_t(vs_data) > UINT32_MAX)) throw std::length_error("uintptr_t(vs_data) > UINT32_MAX");
    if (vs_size > 16777215)                throw std::length_error("vs_size > 16777215");
    if (inicio > 16777215)                 throw std::length_error("inicio > 16777215");
    if (count > 16777215)                  throw std::length_error("count > 16777215");

    bypass_cache_write32(&data_addr->tipo,        float_bits_to_uint32(-tipo));
    bypass_cache_write32(&data_addr->vs_data,     uint32_t(vs_data));
    bypass_cache_write32(&data_addr->vs_size,     float_bits_to_uint32(vs_size*sizeof(T)*CHAR_BIT/8));
    bypass_cache_write32(&data_addr->inicio,      inicio*sizeof(T)*CHAR_BIT/8);
    bypass_cache_write32(&data_addr->count,       float_bits_to_uint32(count));
    bypass_cache_write32(&data_addr->_1,          sizeof(T)*CHAR_BIT/8);
    bypass_cache_write32(&data_addr->_2,          2*sizeof(T)*CHAR_BIT/8);
    bypass_cache_write32(&data_addr->_3,          3*sizeof(T)*CHAR_BIT/8);
    bypass_cache_write32(&data_addr->__1,         -1*sizeof(T)*CHAR_BIT/8);
    bypass_cache_write32(&data_addr->__2,         -2*sizeof(T)*CHAR_BIT/8);
    bypass_cache_write32(&data_addr->use_texture, float_bits_to_uint32(use_texture));
    bypass_cache_write8(control_addr, 0x01);
    if (mode == SYNC) while ((bypass_cache_read8(control_addr)&0x01) != 0x00) ;
}
```

The reason for using templates function is repeated many functions vary only in small areas such as if the vertices texture coordinates contain or not, or the size in bits of the index, in the case of IPLT functions.

As shown in the above code the only thing this function does is to write to the data memory of gpu (which accesses the Nios II) values and the type of processing to be done to fix vertices (if processed the arrangement of vertices and a list of items, a list of lines, a strip line, a list of triangles, one triangle strip or an array of triangles), the address within the data memory in which is stored the array vertices, the size of the array of vertices, the initial vertex, the number of vertices to be processed, the number of bytes that must be added to the address which is stored the array of vertices to move a vertex, 2 vertices, 3 vertices - vertex vertices 1 and -2 and the use or not of texture coordinates.

Finally, the core of the gpu is active, and is the gpu that performs all graphics processing operation from this point, the program is written in assembler is the program memory (this program is loaded only once from one header file when building the gpu structure).

There are two ways the Nios II software through the driver to know when the task is complete graphics processing: a variable mode is when the gpu structure equal to SYNC, which means he will synchronously entering an infinite loop until the core of the gpu is disabled; otherwise the variable mode is when the gpu is equal to ASYNC structure, meaning that it will asynchronously returning the plt function or IPLT immediately and can do other tasks until it receives an interrupt indicating that the task of graphics processing is complete.

Obviously asynchronously is much more efficient because as mentioned above, allows the Nios II do other tasks while the gpu handles the graphics processing, however, worth noting that you cannot change the arrangement of vertices, the transformation matrix or other securities that uses the gpu before you finish the job of graphics processing and display otherwise malfunction.