插值 FPGA 实现分析文档

一、实现要求

功能要求：

对信号 2、5、10、20、25、50、100 倍插值；滤波器系数从外部输入，保存在 RAM 中。

外部输入参数：

312.5M 时钟（插值模块时钟）

100M 时钟（系数存储时钟）

待插值信号（10 bit，10 位 ADC 输出，前级模块发送）

FIFO 满信号（高电平有效，前级模块发送）

输入有效信号（高电平有效，前级模块发送）

复位信号（高电平有效，PC 机发送，地址：0x88B4）

使能信号（高电平有效，PC 机发送，地址：0x88B8）

清零信号（高电平有效，PC 机发送，）

插值倍数（4bit，0 到 6 依次对应 2、5、10、20、25、50、100 倍插值，，PC 机发送，地址：0x88A4）

滤波器系数（17 bit，PC 机发送，高位地址：0x88B0，低 16 位地址：0x88AC）

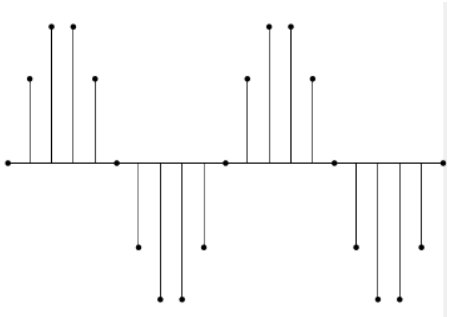
地址（9 bit，PC 机发送，地址：0x88A8）

输入时序：

先产生一个正脉冲复位，复位后同时发送地址和滤波系数，发送滤波系数和地址，等每个地址和滤波系数稳定后，发送一个写使能信号脉冲。在所有的地址和系数发送完成后，将写使能信号变为低电平。发送插值倍数，等清零信号无效后，且在 FIFO 满信号有效后，将带插值的信号输入，同时每发送一个数据需要发送一个输入有效脉冲。

二、插值滤波原理

信号的插值是指增加抽样率以增加数据的过程。将抽样频率为fs的信号$x(n)$进行 L 倍插值，即抽样频率变为Lfs，最简单直接的方式是在$x(n)$每两个点之间插入$L-1$个 0，如图 2-1 所示。记补零后的信号为，则



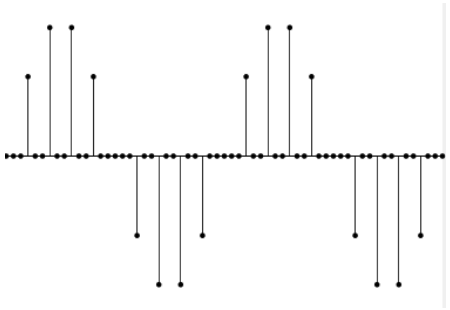


图 2-1　3 倍插值前后图像

$$v(n)=\begin{cases}x(n/L) & n=0,\pm L,\pm 2L,\ldots\\ 0 & 其它\end{cases}$$

从频域分析插值：

对$v(n)$做傅里叶变化有如下关系式

$$V(e^{j\omega})=\sum_{n=-\infty}^{\infty}v(n)e^{-j\omega n}=\sum_{n=-\infty}^{\infty}x(n/L)e^{-j\omega n}=\sum_{k=-\infty}^{\infty}x(k)e^{-j\omega kL}$$

则抽取前后的频域关系为

$$V(e^{j\omega})=X(e^{j\omega L})$$

$$V(z)=X(z^{L})$$

由抽取前后的频域关系可知，$V(e^{j\omega})$是将$X(e^{j\omega L})$频带压缩了 L 倍。$V(e^{j\omega})$和$X(e^{j\omega L})$都是周期的，且$X(e^{j\omega L})$周期为2π，由于$V(e^{j\omega})$是将$X(e^{j\omega L})$频带压缩了 L 倍，则$V(e^{j\omega})$的周期是$2\pi/L$，则在一个周期内产生了$\mathrm{L}-1$个镜像。

插值后，$V(e^{j\omega})$每个2π周期出现了 L 个周期，其中$\mathrm{L}-1$个周期为$V(e^{j\omega})$的映像，需要去除。去除的方法是在$V(e^{j\omega})$之后加一个低通滤波器，以滤除由于频带压缩产生的镜像。低通滤波器的形式如下：

$$H(e^{j\omega})=\begin{cases}L & |w|<\pi/L\\ 0 & 其它\end{cases}$$

低通滤波器的截止频率是π/L。在频域上看，滤波器的主要作用是去除$V(e^{j\omega})$中多余的$\mathrm{L}-1$个映像。

从时域上分析：

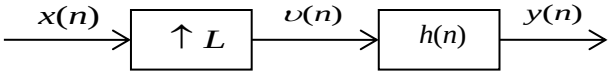


图 2-2 插值后的滤波

如图 2-2 所示，$x(n)$在经过 L 倍 0 插值后，在经过低通滤波器。这时滤波器在时域上表现为对$v(n)$做平滑处理。

输出$y(n)$的时域表达式为

$$y(n) = v(n) * h(n) = \sum_k v(k)h(n-k) = \sum_k x(k/L)h(n-k) \qquad (2-1)$$

或

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-kL) \qquad (2-2)$$

三、插值滤波实现

若将数据插值后再滤波，即$y(n) = v(n) * h(n) = \sum_k v(k)h(n-k)$，由于插值后$v(n)$有很多零值，而零值与$h(n)$相乘为零，这实际上是不需要的；在实现时，乘零运算也会占用乘法器的资源，由于乘法器的资源有限，所以要避免不必要的乘零运算。

对 2-2 式的表达形式，以三倍插值为例，滤波器系数个数为 12，则有

$$y(9) = \sum_{k=-\infty}^{\infty} x(k)h(9-3k) = \sum_{m=0}^{3} x(3-m)h(9-3k))$$

$$= h(0)x(3) + h(3)x(2) + h(6)x(1) + h(9)x(0)$$

同理可得

$$y(9) = h(0)x(3) + h(3)x(2) + h(6)x(1) + h(9)x(0)$$
$$y(10) = h(1)x(3) + h(4)x(2) + h(7)x(1) + h(10)x(0)$$
$$y(11) = h(2)x(3) + h(5)x(2) + h(8)x(1) + h(11)x(0)$$
$$y(12) = h(0)x(4) + h(3)x(3) + h(6)x(2) + h(9)x(1)$$
$$y(13) = h(1)x(4) + h(4)x(3) + h(7)x(2) + h(10)x(1)$$
$$y(14) = h(2)x(4) + h(5)x(3) + h(8)x(2) + h(11)x(1)$$

根据以上的计算过程，可以发现，每输入一个数据$x(n)$，要产生 L 个输出$y(\mathrm{L}n)$、$y(\mathrm{L}n-1)$、$y(\mathrm{L}n-2)$、…、$y(\mathrm{L}n-\mathrm{L}-1)$。如图 3-1 所示，根据式 2-2 的表达式，可以画出如下的框图完成对插值的滤波实现。

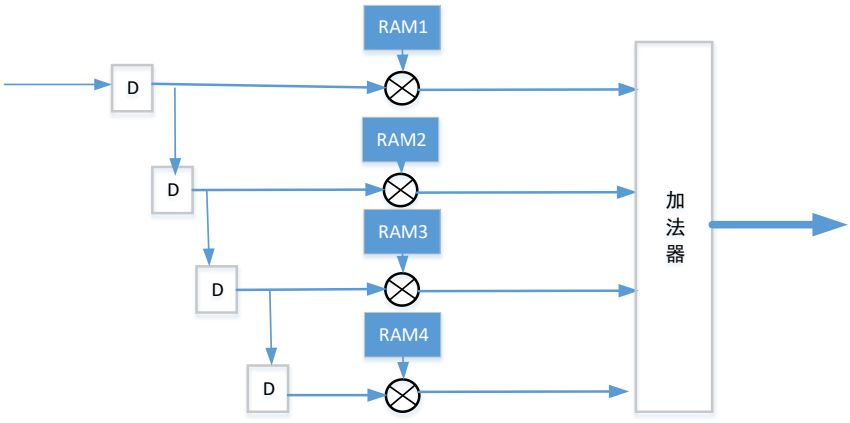


图 3-1 实现框图

RAM1 中依次存储的是$h(0)$、$h(1)$、$h(2)\ldots h(L-1)$，RAM2 中依次存储的是$h(L+0)$、$h(L+1)$、$h(L+2)\ldots h(2L-1)$，直到存完所有的 RAM。

四、FPGA 实现流程

如图 4-1 所示，为插值的滤波器实现框图。将插值实现分两个过程实现：存滤波器系数过程和插值滤波过程。

存滤波器系数过程：由于插值要最大实现 100 倍插值，则每个 RAM 中最少要存储 100 个滤波系数，本次使用 30 个 RAM，每个 RAM 中存储 100 个滤波系数。插值的倍数可以选择，所以对外部发送的滤波器系数要做处理再发送到插值模块中，所以在对滤波系数存储时，要按照一定的规律存储。

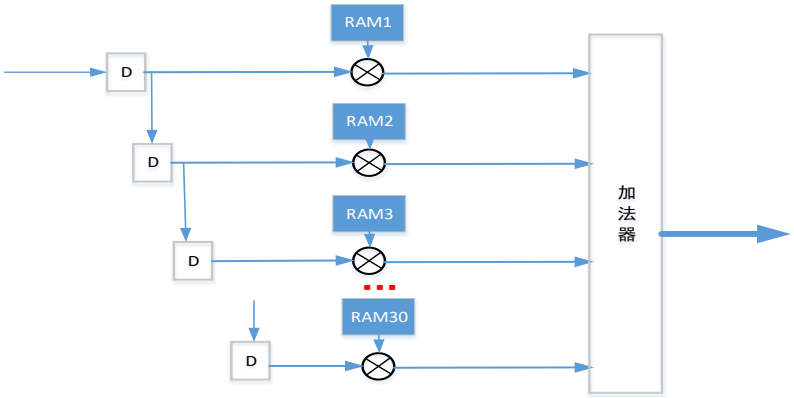


图 4-1 FPGA 实现框图

存滤波器系数过程是将 3000 个系数写入 30 个 RAM 中，每个 RAM 中存 100 个数。对于第 i 个 RAM，需要存入的滤波器系数对应的地址依次为 30n+i，n=0,1,2，…，99。每个 RAM 的地址都是从 0 到 99。

在存滤波系数时，第一步是产生每个 RAM 的写地址和写使能，第二步是将滤波系数写入 RAM 中，所以我们可以建立两个模块实现其各自的功能。

插值滤波过程：在滤波器系数全部存入所有 RAM 后，按照插值原理，将待插值信号输入，通过控制时序，使待插值信号和滤波系数在时序上对齐，并将乘法结果相加得到输出的过程。这个过程在 dpx_clk 时钟下工作。在插值过程中，需要发送 RAM 读地址以得到滤波系数。

在插值滤波过程中，第一步要根据插值倍数来选择不同的地址输出，以读取 RAM 中对应的滤波系数，并将输出的滤波系数与输入信号在时序上对齐，第二步是要将滤波系数和输入信号相乘，并将所有的乘运算结果输入到加法器并输出。此过程需要设计两个模块来设计实现其功能。

五、模块分析

顶层模块的主要作用是调用底层的模块以实现功能要求。插值的实现中，使用的顶层模块的声明如下：

```
module Various_inter(
input              dpx_clk          ,// 时钟信号
input              adsp_clk         ,// 时钟信号
input              fifo_clr         ,// 清零
input              valid_in         ,//有效输入信号
input    [9:0]     data_in          ,// 待插值信号
input              adc_fifo_full    ,// FIFO 满信号
input    [3:0]     sample_mode      ,// 插值倍数选择
```

```
input     [9:0]     mult_factor_wa    , //写地址
input     [16:0]    mult_factor_wd    ,//写数据
input               factor_w_rst      ,//复位信号
input               mult_factor_wen ,//写使能信号

output              rd_fifo           ,
output    reg       valid_out         , //有效数据，高电平有效

output    [9:0]     data2fifo               //数据输出
);
```

顶层模块主要调用两个模块：interpolation_first 和 interpolation 模块。对 interpolation_first 模块只调用一次，而对 interpolation 模块调用了 29 次。这两个模块的功能基本相似，将滤波器系数存入各个实例中，将待插值信号传递到下一级，同时将运算结果传递到到下一级。如图 5-1 所示为顶层调用的的框图。

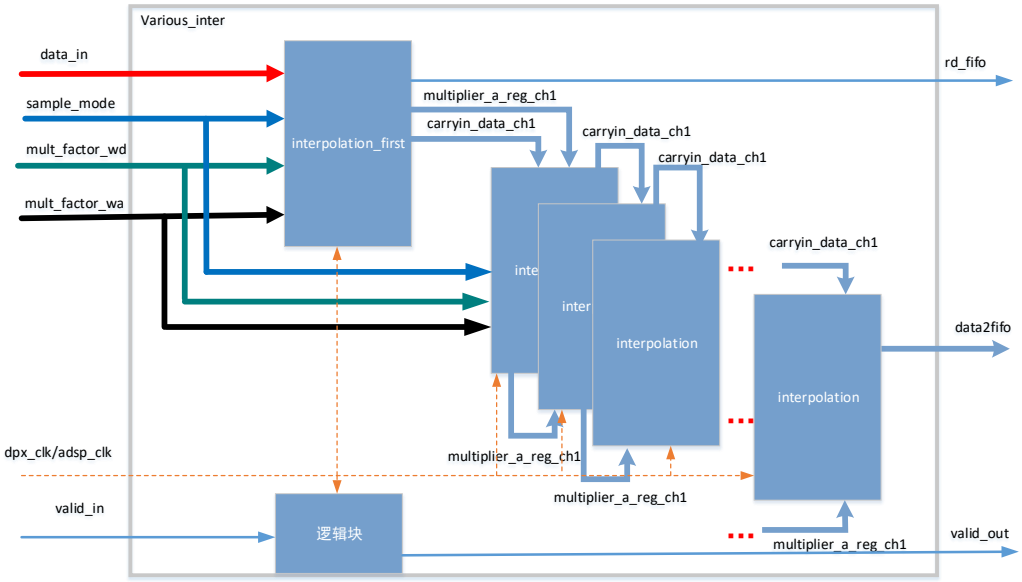


图 5-1 顶层调用的框图（只标注了主要信号流向）

```
always @ ( posedge dpx_clk )
begin
    if ( fifo_clr )
    begin
        cnt       <= 0 ;
        valid     <= 0 ;
    end
    else if ( cnt == 30 )  //
    begin
        cnt       <= cnt ;
        valid     <= 1   ;
    end
    else if ( valid_in == 1 )
    begin
        cnt <= cnt + 1 ;
    end
    else ;
end
```

```
always @ ( posedge dpx_clk )
begin
    if ( valid == 0 )
    begin
        valid_cnt     <= 0 ;
        valid_out     <= 0 ;
    end
    else if ( valid_cnt == 38 )
    begin
        valid_cnt     <= valid_cnt ;
        valid_out     <= 1   ;
    end
    else
    begin
        valid_cnt <= valid_cnt + 1 ;
    end
end
```

如上图中的代码，当输入 30 个有效输入时，根据公式 2-2，就会输出第一个有效数据，但是由于代码设计中会对数据延时，导致第一个有效输出会延迟

输出（代码中的固定延时是 38 个延时单位）。

2、interpolation_first/interpolation 模块

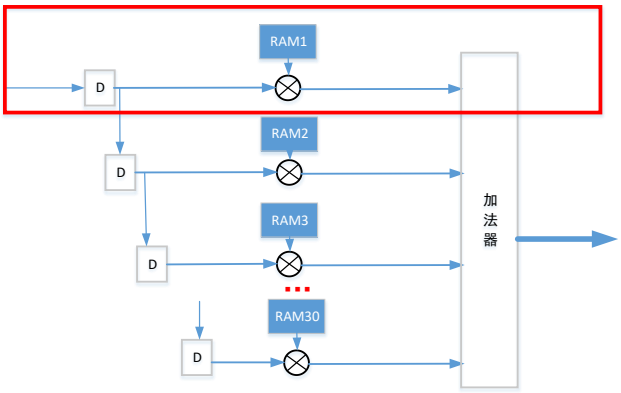


图 5-2 FPGA 实现框图

interpolation_first/interpolation 模块相当于实现图 5-2 中红色框内标注的功能。因为需要存入 30 个 RAM，所以需要调用 interpolation_first/interpolation 模块共三十次。

interpolation_first 模块主要调用了四个模块：Factor_addr_gen_int、RAM_interpolate、data_process_int_first、data_operation。interpolation 模块主要调用了四个模块：Factor_addr_gen_int、RAM_interpolate、data_process_int、data_operation。两个模块只是 data_process_int_first 和 data_process_int 的区别，但是其功能是相似的。如图 5-3 所示,为各个模块的数据流向，各个模块的功能主要如下：Factor_addr_gen_int 模块是通过外部发送来的写地址和写使能，来产生 RAM 模块的写地址 mult_factor_wa 和写使能 ram_wen，并与 mult_factor_wd 一起输入到 RAM_interpolate 模块的 a 端口（负责数据的写入，使数据存储到 RAM 中），这个过程是在 dsp_clk 时钟下工作的。RAM 模块的 b 端口将输入的地址 addrb 对应 RAM 中的值 doutb 输出到 data_process_int_first/data_process_int 模块中，data_process_int_first/data_process_int 模块根据抽取倍数选择不同的地址输出，使 RAM 中输出不同的滤波器系数，同时将输入信号 data_in_ch1 输出到 multiplier_a_reg_ch1，并负责将系数 multiplier_b 与 multiplier_a 在时序上对齐；data_operation 模块将输入的 multiplier_b 与 multiplier_a 相乘，并与输入的 carryin_data_ch1 相加，这个过程是在 dpx_clk 时钟下工作的。

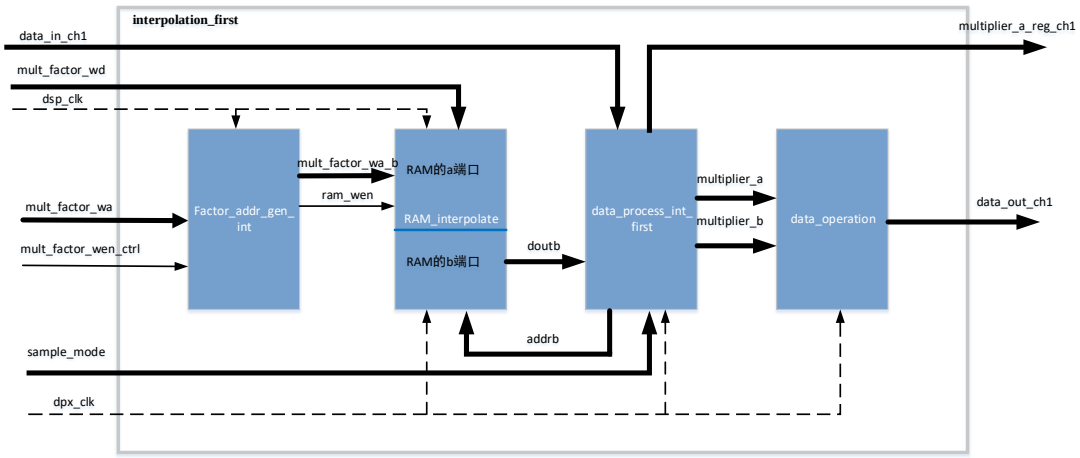


图 5-3 interpolation_first 模块框图

interpolation_first 模块端口说明如下：

```
module interpolation_first(
input                    dpx_clk                 , // 时钟
input                    dsp_clk                 , // 时钟
input                    fifo_clr                , // 清零端
input                    fifo_full               , //FIFO 满
input                    valid_in                ,//有效输入
input    [2:0]           sample_mode             , // 插值倍数选择
input                    mult_factor_wen_ctrl , //写使能信号
input    [9:0]           mult_factor_wa          , // 地址
input    [9:0]           data_in_ch1             , // 输入待插值信号
input    [16:0]          mult_factor_wd          , // 滤波器系数
input                    factor_w_rst            , // 写复位信号
input    [5:0]           i                       , // 片选
input    [32:0]          carryin_data_ch1        , // 数据输入
output   [32:0]          data_out_ch1            , // 数据输出
output                   rd_fifo                 ,
output   [9:0]           multiplier_a_reg_ch1    // 数据输出
);
```

interpolation 模块只是未定义输出端口 rd_fifo。

2.1 Factor_addr_gen_int 模块

1）端口说明

```
module Factor_addr_gen_int(
input                    clk                     , // 时钟
input                    factor_w_rst            , // 复位信号，高有效
input                    mult_factor_wen_ctrl , // 写使能信号，高有效
input          [9:0]     mult_factor_wa          , //地址
input          [5:0]     ram_wen_sel_cnt         , //片选
```

output reg [6:0] mult_factor_wa_b , // RAM 写地址
output reg ram_wen // RAM 写使能信号
);

2）功能描述：

产生 RAM 写地址和写使能。

在 factor_w_rst 和 mult_factor_wen_ctrl 都有效时，通过 RAM 片选信号 ram_wen_sel_cnt 和输入地址 mult_factor_wa，在输出端输出 RAM 写使能 ram_wen 和 RAM 写地址 mult_factor_wa_b。将 3000 个系数中的第 0、30、60、…、2970 个系数存入 RAM1，第 1、31、61、…、2971 个系数存入 RAM2，依次类推。

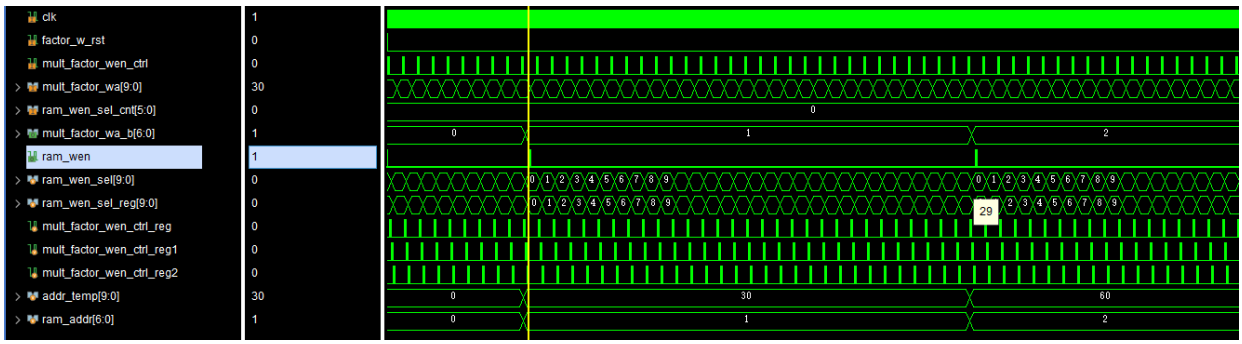
3）实现思路及代码分析

地址 mult_factor_wa 从 0 开始不断加 1， mult_factor_wa 值改变一次都发送一个 mult_factor_wen_ctrl 写使能。mult_factor_wa 以 30 个数为一个周期生成一个地址并输出，并通过片选信号 ram_wen_sel_cnt 选择对应模块 ram_wen 输出有效。（共例化了 30 个 Factor_addr_gen 模块，片选信号 ram_wen_sel_cnt 分别为从 0 到 29）。

```
always @ (posedge clk)
begin
    if(factor_w_rst)
    begin
        addr_temp <= 10'h00;
        ram_addr <= 7'h00;
    end
    else if(ram_wen_sel==10'h01D && (mult_factor_wen_ctrl==1'b1))/,
    begin
        ram_addr <= ram_addr + 1'b1;   //写完30个ram才加1
        addr_temp <= addr_temp + 6'h1E;// 一共30个ram，依次读取ram'
    end
      else
      begin
            ram_addr <= ram_addr;
            addr_temp <= addr_temp;
      end
end
```

```
always @ (posedge clk)
begin
    if(factor_w_rst)
    begin
        ram_wen_sel <= 10'h0;
    end
    else if(mult_factor_wen_ctrl_reg==1'b1)
        begin
            ram_wen_sel <= mult_factor_wa - addr_temp;
        end
end
```

根据上图中的代码，mult_factor_wa 由外部输入且不断加 1，随着 mult_factor_wa 的增大，片选信号 ram_wen_sel 不断增大直到 29，在 ram_wen_sel 增大的过程中，对应的不同 RAM 被选中，但是输出的地址保持不变；当 ram_wen_sel=29 时，ram_addr 地址的值加 1，addr_temp 的值加 30，这样在 mult_factor_wa 继续增大的过程中，ram_wen_sel 的值总保持在 0 到 29 之间，使得写使能 ram_wen 在 30 个 RAM 之间来回切换。如下图所示。



2.2 RAM_interpolate 模块（RAM IP 核）

1）端口说明

RAM_interpolate factor_inst (

```
    .clka(dsp_clk),          // RAM 写时钟
    .ena(1),                 // RAM 使能，置高
    .wea(ram_wen),           // RAM 写使能，高有效
    .addra(mult_factor_wa_b),      //（input）RAM 写地址
    .dina(mult_factor_wd),     //  （input）滤波器系数
    .clkb(dpx_clk),          // RAM 读时钟
    .enb ( 1'b1)             //RAM 读使能，置高
    .addrb(mult_factor_ra_b),  // RAM 读地址
    .doutb(mult_factor_rd_b)   //  滤波器系数输出
);
```

2）功能描述：

将滤波系数写入 RAM 中，并将对应地址的滤波系数输出。

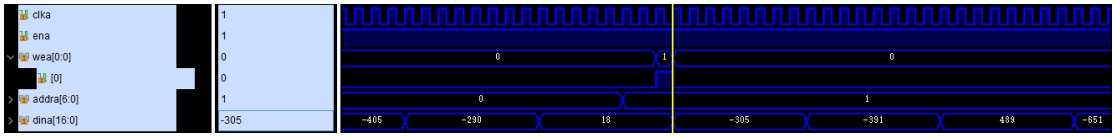
RAM 写数据（a 端口）：当 ram_wen 和 ena 都有效时，以 dsp_clk 为时钟信号，将数据 mult_factor_wd 写入 RAM 的地址 mult_factor_wa_b 中。

RAM 读数据（b 端口）：在 dpx_clk 时序下，将地址 mult_factor_ra_b 对应的数据发送到 mult_factor_rd_b 端。

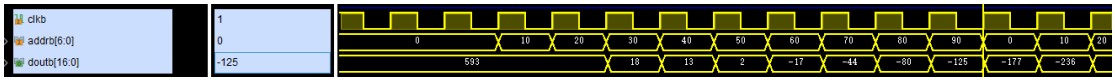
3）实现思路及代码分析

调用 RAM 的 IP 核

RAM 写：在 Factor_addr_gen 模块的地址输出端和写使能的控制下，将滤波器系数存在 RAM 中。在 RAM 写使能有效时，要确保写地址和滤波器系数输入数据时序正确，否则将导致存入的数据发生移位，对插值结果产生影响。可以通过控制 Factor_addr_gen 模块写使能输出的延迟来调节时序。



RAM 读：从读地址输入到数据输出，存在时间延时（图中为两个单位的延时）。



2.3 data_process_int_first / data_process_int 模块

1）端口说明

```
module  interpolation_first (
input                   dpx_clk,     // 时钟
input                   fifo_clr,    // 清零，高有效
input                   fifo_full,   // fifo 满信号，高有效
input                   valid_in ,   //输入有效，高有效
input     [9:0]         data_in,     // 数据输入（待插值信号）
input     [2:0]         sample_mode,   // 插值倍数
input [16:0]  mult_factor_rd_b,        // 滤波器系数
//output
output reg [6:0]        mult_factor_ra_b_out,    // RAM 读地址
```

```
output reg [9:0]        multiplier_a,     //待插值信号
output reg [16:0]       multiplier_b,     //滤波器系数
output reg                    rd_fifo,    //
//待插值信号
output reg [9:0]        multiplier_a_reg
);
```

在端口声明中，data_process_int 中只少了一个输出端口 rd_fifo。

2）功能描述：

产生 RAM 读地址，并将滤波器系数和输入信号输出到乘法器，同时将输入信号输出到下个 interpolation 模块。

在 fifo_full 有效且 fifo_clr 无效时，通过插值倍数 sample_mode 的选择，控制 RAM 的读地址 mult_factor_ra_b_out 输出，将输入的滤波器系数 mult_factor_rd_b 和待插值信号 data_in 通过时序控制使输出的 multiplier_a 和 multiplier_b 数据对齐，同时在 valid_in 有效时将 data_in 输出到 multiplier_a_reg 端口。

3）实现思路及代码分析

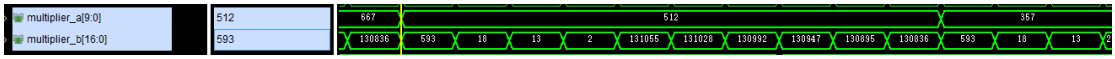
```
always @ ( posedge dpx_clk )
begin
    case ( sample_mode )
        3'b000:
        begin
            interp_mul      <=   7'h02 ;//2倍
            base_parameter  <=   7'h32 ;
        end
        3'b001:
        begin
            interp_mul      <=   7'h05 ;//5倍
            base_parameter  <=   7'h14 ;
        end
        3'b010:
        begin
            interp_mul      <=   7'h0a ;//10倍
            base_parameter  <=   7'h0a ;
        end
        3'b011:
        begin
            interp_mul      <=   7'h14 ;//20倍
            base_parameter  <=   7'h05 ;
        end
        3'b100:
        begin
            interp_mul      <=   7'h19 ;//25倍?
            base_parameter  <=   7'h04 ;
        end
        3'b101:
        begin
            interp_mul      <=   7'h32 ;//50倍
            base_parameter  <=   7'h02 ;
        end
        3'b110:
        begin
            interp_mul      <=   7'h64 ;//100倍
            base_parameter  <=   7'h01 ;
        end
        default:
        begin
            interp_mul      <=   7'h64 ;//100倍
            base_parameter  <=   7'h01 ;
        end
    endcase
end
```

1）对于不同的插值倍数，控制 interpolation 模块输出不同的地址 mult_factor_ra_b_out，并读出 RAM 对应地址中滤波系数；如上图所示，base_parameter 为对应插值倍数下的地址变化间隔。以 10 倍插值为例，读取每个 RAM 中第 0、10、20、…、90 个地址对应的滤波器系数。

2）控制时序将输出的待插值信号和滤波器系数对齐，输出到乘法器模块。

```
always @ ( posedge dpx_clk )
begin
    if ( reset == 1 )
    begin
        mult_factor_ra_b_out <=   7'h0 ;
        cnt_factor           <=   7'b0 ;
    end
    else if ( cnt_factor == (interp_mul - 7'h01) || valid_in == 1'b1 )
    begin
        cnt_factor               <=   7'h0 ;
        mult_factor_ra_b_out[6:0] <=   7'h0 ;
    end
    else
    begin
        mult_factor_ra_b_out[6:0] <=   mult_factor_ra_b_out[6:0] + base_parameter ;
        cnt_factor               <=   cnt_factor + 1'b1 ;
    end
end
```

以 10 倍插值为例，cnt_factor 每计数一次，输出到 RAM 读地址改变一次，RAM 读地址为第 0、10、20、…、90，并从 RAM 中取出对应地址所保存的滤波器系数输出到 multiplier_b。



当 multiplier_a 端的待插值信号与 multiplier_b 的系数先后时序不对应时（每个 multiplier_a 数据要与 0 地址对应的系数对齐，如上图所示，593 是 RAM1 中 0 地址存储的滤波器系数），可以通过对 multiplier_a 端增删延时来达到时序和数据对应，即修改下图所示的 ram 读数延时代码段。

```
always @ ( posedge dpx_clk )      //ram 读数延迟
begin
    multiplier_a_reg1 <=  multiplier_a_reg  ;
    multiplier_a_reg2 <=  multiplier_a_reg1 ;
    multiplier_a_reg3 <=  multiplier_a_reg2 ;
    multiplier_a      <=  multiplier_a_reg3 ;
end
```

3) data_process_int_first 和 data_process_int 模块的区别。rd_fifo 是 data_process_int_first 中多定义的输出端口，且 rd_fifo 的使用仅在如下图所示的代码中，未被其他模块调用。

```
always @ ( posedge dpx_clk )
begin
    if ( reset == 1 )
    begin
        cnt     <=  7'b0 ;
        rd_fifo <=  1'b0 ;
    end
    else if ( cnt == (interp_mul - 7'h01) )
    begin
        cnt     <=  7'h0 ;
        rd_fifo <=  1'b1 ;
    end
    else
    begin
        cnt     <=  cnt + 1'b1 ;
        rd_fifo <=  1'b0 ;
    end
end
```

2.4 data_operation 模块

1）端口说明

```
module data_operation (
input                dpx_clk,     // 时钟
input                fifo_clr,    // 清零
input                fifo_full,   //  fifo 满信号
input    [9:0]       multiplier_a,    // 乘法器输入（待插值信号）
input    [16:0]      multiplier_b,    // 乘法器输入（滤波器系数）
input    [32:0]      carryin_data,    // 加法器输入
input    [5:0]       delay_cnt,       // 延时
output   [32:0]      data_out         //数据输出（乘加结果）
    );
```

2）功能描述：

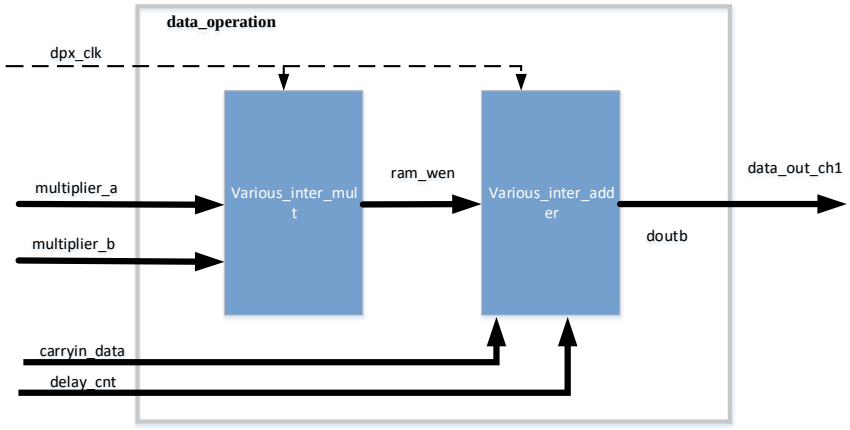


图 5-4 data_operation 调用模块框图

实现乘法和加法运算。如图 5-4 所示，data_operation 调用了 Various_inter_mult 和 Various_inter_adder 两个模块。

在 fifo_full 有效且 fifo_clr 无效时，乘加器在 dpx_clk 时钟下工作，将 multiplier_a 与 multiplier_b 相乘的运算结果延时 delay_cnt 个单位后，与 carryin_data 相加输出到 data_out。

3）实现思路及代码分析

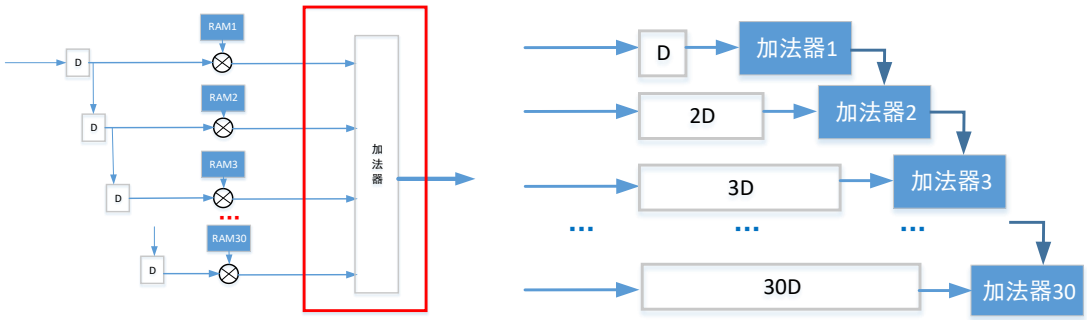


图 5-5 加法器的 FPGA 实现

注：图中的 D 表示延时，D 前的数字表示延时的个数

将待插值信号和滤波器系数输入到乘法器后，经过固定的延时得到乘法运算结果，由于编程时实现对代码的复用，采用菊花链式的编程方式。所以利用图 5-5 右来实现左边所示的加法器，将乘法运算的结果分别延时不同的时间，再经过加法器。这样在时钟有效沿时，乘法结果恰好与上一个加法器的输出相加，提高了代码的复用性。

2.4.1 Various_inter_mult（乘法器 IP 核）

1）端口说明

```
Various_inter_mult mult_ins (
    .clk(dpx_clk),                  //时钟
    .a(multiplier_a_temp[10:0]),    //乘法器输入
    .b(multiplier_b[16:0]),         //乘法器输入（滤波器系数）
    .ce(fifo_full_temp),            //控制，高有效
    .p(m_d)                         //乘法结果输出
);
```

2）功能描述：

实现乘法运算。

在 fifo_full_temp 有效时，乘法器在时序作用下工作，将 multiplier_a_temp 与 multiplier_b 相乘输出到 m_d 端。

3）实现思路及代码分析

乘法器 IP 核调用

2.4.2 Various_inter_adder

1）端口说明

```
module Various_inter_adder (
input               clk,          //时钟
input               fifo_clr,     //清零，高有效
input               fifo_full,    //fifo 满信号，高有效
input     [27:0]    d_in,         //加法器输入（乘法器输出）
input     [32:0]    carryin_data,     //加法器输入
input     [5:0]          delay_cnt,    //延时时间

output reg [32:0] data_out          //加法结果
                );
```

2）功能描述

实现延时和加法运算。

在 fifo_full 有效且 fifo_clr 无效时，加法器才开始工作，d_in 在延时 delay_cnt 个单位后，与 carryin_data 相加并输出结果 data_out。

3）实现思路及代码分析

乘法器同时输出了乘法结果采用图 5-5 所示的流程编程，利用 30 个 32 位的加法器实现加法。

延迟的实现代码如下：

```
genvar i;
generate
for(i = 0;i < 28;i = i + 1)
begin: loop1
    SRLC32E #(
      .INIT(32'h00000000) // Initial Value of Shift Register
    ) SRLC32E_inst (
      .Q(d_in_aft_srl[i]),     // SRL data output
      .Q31(), // SRL cascade output pin
      .A(delay_cnt),     // 5-bit shift depth select input
      .CE(1'b1),   // Clock enable input
      .CLK(clk), // Clock input
      .D(d_in[i])      // SRL data input
   );
end
endgenerate
```

通过移位寄存器（SRL32E）实现延迟的结果。通过 delay_cnt 控制 d_in 延时 1 到 30 个时间单位（延时时间是 delay_cnt+1）。