# 多模式乘法器 RTL 设计

一、设计要求

**设计一款可配置多功能运算单元，输入两个 32bit 操作数 A, B，输出 64bit 操作数 Mout。该乘法单元根据不同操作码可以完成如下操作**

**功能 1：单纯乘累加 Mout = Mout + A * B（A,B 均为有符号数，补码）**

**功能 2：4 个 8*8 像素点积累加**

（1）点积求和

Sum[15:0] = (A[31:24])*(B[31:24]) + (A[23:16])*(B[23:16])

+ (A[15:8])*(B[15:8]) +(A[7:0])*(B[7:0]);

（2）根据控制码选择存放位置

case (ww)

11：Mout[63:48] = Mout[63:48] + Sum;

10：Mout[47:32] = Mout[47:32] + Sum;

01：Mout[31:16] = Mout[31:16] + Sum;

00：Mout[15:0] = Mout[15:0] + Sum;

**操作如有溢出自动求饱和。**

**功能 3： 16 位复数条件控制乘累加减**

Mout[63:32] = Mout[63:32] + $(-1)^{ctrl}$ * (A[31:16] * B[15:0]+

A[15:0] * B[31:16]);

Mout[31: 0] = Mout[31:0] + (-1)$^{ctrl}$ * (A[15:0] * B[15:0]-A[31:16] * B[31:16]);

说明：A 和 B 的高 16 位和低 16 位分别做复数的虚部和实部，做 16 位复数乘法。 复数乘法结果根据**模式位 ctrl 情况**和 Mout 指定的累加器所存的复数进行累加或累减。 最终结果复数实部的 32 位结果放在 Mout 低 32 位，复数虚部的 32 位结果放在 Mout 高 32 位。

**如果有上下溢出，自动求饱和，并给出溢出信号**

**溢出运算的规则为：**

**上溢：取正数最大值 7fff（根据实际位宽调整）**

**下溢：取负数最小值 1000（根据实际位宽调整）**

**要求：**

- **以 8*8 乘法器为基本单元，尽量复用运算单元**
- **模块控制码及流水线划分可自行定义，如果无法实现一拍一个运算的完全流水，可以添加必要的状态控制逻辑实现多拍一个运算流程**

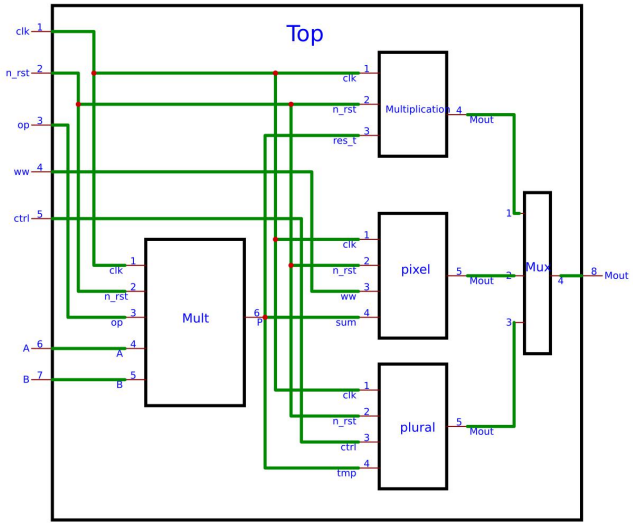
## 二、设计思路

根据要求，首先明确整个设计的输入输出。这里设置输入 clk 为整个电路提供基准时钟，设置输入 n_rst 为整个电路提供低电平复位信号。因为多模式乘法

器由三个功能组成，因此设置功能选择输入 op，其位宽设置为 3 位，当 op=3’b001 时激活功能一，当 op=3’b010 时激活功能二，当 op=3’b100 时激活功能三。根据 8*8 像素点积累加功能设置位宽为 2 位的控制码 ww。根据功能三设置位宽为一位的控制码 ctrl。

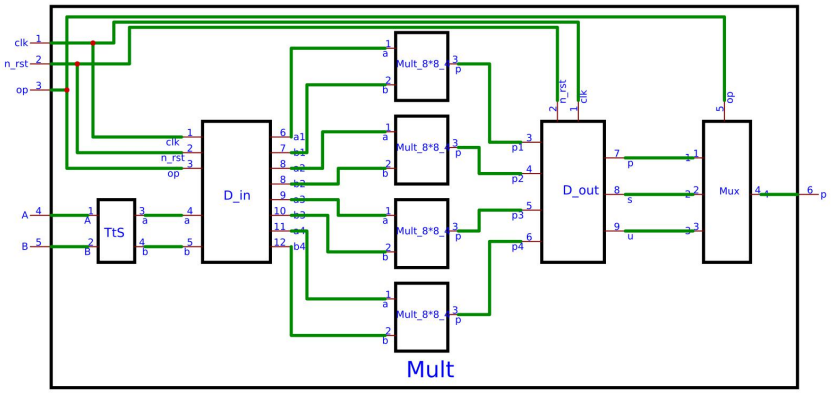
多模式乘法器的输入数据分别为 A、B，其位宽 32 位。对于功能一，A、B 输入 32 位补码数，功能二输入由 4 个 8 位补码数拼接而成的 32 位数，功能三则输入由两个 16 位补码数拼接而成的 32 位数。多模式乘法器最终计算结果为 Mout，其位宽 64 位，由 op 控制其输出的结果为功能一或功能二或功能三。

此多模式乘法器可做如下设计。首先根据功能选择输入 op 对输入数据 A、B 做预处理，即对应功能一，直接将 A、B 两个 32 位补码数转换为其对应的 32 位原码数；对应功能二，现将 A、B 中的 4 个 8 位补码数提取出来，然后再转换为对应的 8 位原码数；对应功能三，现将 A、B 中的 2 个 16 位补码数提取出来，然后再转换为对应的 8 位原码数。然后根据功能选择输入 op 将处理好的数据去掉符号位，将符号位统一置零后输入乘法器(对于乘法器的具体设计将在下文介绍)进行计算，将计算结果转换成补码后输入到具体的功能模块执行下一步运算。具体功能模块运算完成后将结果输入 mux 由 op 三选一，将最终结果输出。多模式乘法器整体功能设计电路如图一所示，模块文件为 top.v：



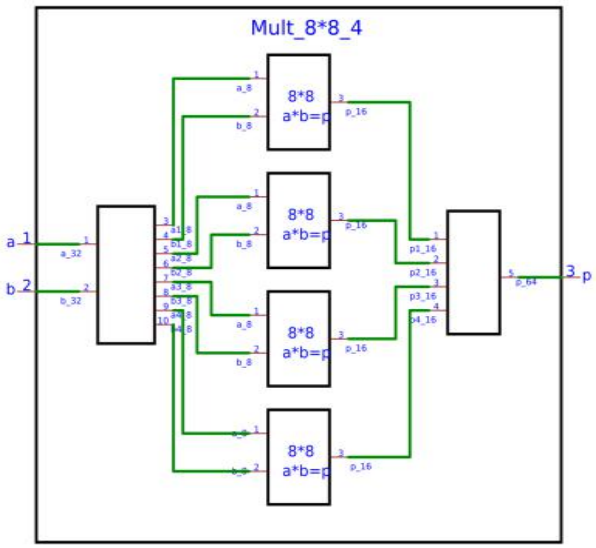

**图一：多模式乘法器 top 模块电路**

## 三、乘法器设计




**图二：乘法器电路模块**

乘法器设计电路如图二所示，模块文件为 mult.v。此乘法器通过复用 4 个由 4 个 8*8 乘法器组成的 8*8_4 乘法器（见图三，即复用 16 个基本的 8*8 乘法器），通过功能选择 op 控制数据输入来完成三个功能的乘法操作。




**图三：8*8_4 乘法器电路模块**

功能一中首先将输入的 32 位补码数字 A、B 转换成原码，并将符号位置零，记为 a、b，此功能对应图二中 Tts 模块。然后开始组织输入 4 个 8*8_4 乘法器的数据，输入这 4 个乘法器的乘数 a1、a2、a3、a4 均为 32 位的 a，而 b 的 0-7 位扩展 4 次组成 32 位的 b1，b 的 8-15 位扩展 4 次组成 32 位的 b2，b 的 16-23 位扩展 4 次组成 32 位的 b3，b 的 24-31 位扩展 4 次组成 32 位的 b4，此功能对应 D_in 模块。接着组织乘法器结果输出，将运算结果转换成补码输出，此功能对应 D_out 模块：

```
tmp_p1 = {{24{1'b0}},p1[15:0]} + {{16{1'b0}},p1[31:16],{8{1'b0}}} + {{8{1'b0}},p1[47:32],{16{1'b0}}} + {p1[63:48],{24{1'b0}}};
tmp_p2 = {{24{1'b0}},p2[15:0]} + {{16{1'b0}},p2[31:16],{8{1'b0}}} + {{8{1'b0}},p2[47:32],{16{1'b0}}} + {p2 [63:48],{24{1'b0}}};
tmp_p3 = {{24{1'b0}},p3[15:0]} + {{16{1'b0}},p3[31:16],{8{1'b0}}} + {{8{1'b0}},p3[47:32],{16{1'b0}}} + {p3[63:48],{24{1'b0}}};
tmp_p4 = {{24{1'b0}},p4[15:0]} + {{16{1'b0}},p4[31:16],{8{1'b0}}} + {{8{1'b0}},p4[47:32],{16{1'b0}}} + {p4[63:48],{24{1'b0}}};
tmp_pt = {{24{1'b0}},tmp_p1} + {{16{1'b0}},tmp_p2,{8{1'b0}}} + {{8{1'b0}},tmp_p3,{16{1'b0}}} + {tmp_p4,{24{1'b0}}};
p = a[DW-1]^b[DW-1] ? {1'b1, ~tmp_pt[2*DW-2:0]+1'b1} : tmp_pt;
```

以上 32*32 乘法原理为：

AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA

```
                                                x        AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
                                                ---------------------------------------------------------------------
                                              AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
                                            AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
                                          AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
                                      +   AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
    ---------------------------------------------------------------------------------------------------------------------------------------------
                                        AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
```

以上 32*8 乘法原理为：

```
                                                                AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
                                                              x                                   AAAAAAAA
                                                              ---------------------------------------------------------------------
                                                                                    AAAAAAAA AAAAAAAA
                                                                              AAAAAAAA AAAAAAAA
                                                                          AAAAAAAA AAAAAAAA
                                                                +         AAAAAAAA AAAAAAAA
                                                              ---------------------------------------------------------------------------------
                                                                AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
```

功能二中首先将输入的 32 位数字先分离成 4 个 8 位的补码数字，然后将 8 位补码数字转换为原码，并将符号位置零，记为 a_1，a_2，a_3，a_4/b_1，b_2，b_3，b_4。然后将 4 个 8 位的 a_1，a_2，a_3，a_4 分别作为 32 位 a1 的 0-7，8-15，16-23，24-31 位，4 个 8 位的 b_1，b_2，b_3，b_4 分别作为 32 位 b1 的 0-7，8-15，16-23，24-31 位。将 a1、b1 输入第一个 8*8_4 乘法器，因此功能二只使用了 4 个 8*8 的基本乘法器。对第一个 8*8_4 乘法器的输出 p1，其 0-15,16-31,32-47,48-63 位分别位 a_1*b_1，a_2*b_2，a_3*b_3，a_4*b_4 的结果，将结果分离后转换为补码，再将四个结果相加即为 Sum，对应 D_out 模块输出 s。

以上 8*8 乘法原理为：

```
                                                                AAAAAAAA
                                                              x AAAAAAAA
                                                              ----------
                                                            AAAAAAAA AAAAAAAA
```

功能三中首先将输入的 32 位数字先分离成 2 个 16 位的补码数字，然后将 16 位补码数字转换为原码，并将符号位置零，记为 ca1，ca2/cb1，cb2。然后按照功能要求将数据组织为：

```
a1 = {{1'b0,ca2[14:0]}, {1'b0,ca2[14:0]}};
a2 = {{1'b0,ca1[14:0]}, {1'b0,ca1[14:0]}};
a3 = {{1'b0,ca1[14:0]}, {1'b0,ca1[14:0]}};
a4 = {{1'b0,ca2[14:0]}, {1'b0,ca2[14:0]}};
b1 = {{1'b0,cb1[14:8]}, {1'b0,cb1[14:8]}, cb1[7:0], cb1[7:0]};
b2 = {{1'b0,cb2[14:8]}, {1'b0,cb2[14:8]}, cb2[7:0], cb2[7:0]};
b3 = {{1'b0,cb1[14:8]}, {1'b0,cb1[14:8]}, cb1[7:0], cb1[7:0]};
b4 = {{1'b0,cb2[14:8]}, {1'b0,cb2[14:8]}, cb2[7:0], cb2[7:0]};
```

然后输入乘法器运算，最终运算结果组织为：

```
tmp_u1 = {{16{1'b0}},p1[15:0]} + {{8{1'b0}},p1[31:16],{8{1'b0}}} + {{8{1'b0}},p1[47:32],{8{1'b0}}} + {p1[63:48],{16{1'b0}}};
tmp_u2 = {{16{1'b0}},p2[15:0]} + {{8{1'b0}},p2[31:16],{8{1'b0}}} + {{8{1'b0}},p2[47:32],{8{1'b0}}} + {p2[63:48],{16{1'b0}}};
tmp_u3 = {{16{1'b0}},p3[15:0]} + {{8{1'b0}},p3[31:16],{8{1'b0}}} + {{8{1'b0}},p3[47:32],{8{1'b0}}} + {p3[63:48],{16{1'b0}}};
tmp_u4 = {{16{1'b0}},p4[15:0]} + {{8{1'b0}},p4[31:16],{8{1'b0}}} + {{8{1'b0}},p4[47:32],{8{1'b0}}} + {p4[63:48],{16{1'b0}}};
cc1 = ca2[15]^cb1[15] ? {1'b1, ~tmp_u1[30:0]+1'b1} : tmp_u1;
cc2 = ca1[15]^cb2[15] ? {1'b1, ~tmp_u2[30:0]+1'b1} : tmp_u2;
cc3 = ca1[15]^cb1[15] ? {1'b1, ~tmp_u3[30:0]+1'b1} : tmp_u3;
cc4 = ca2[15]^cb2[15] ? tmp_u4 : {1'b1, ~tmp_u4[30:0]+1'b1};
tmp_u5 = cc1 + cc2;
tmp_u6 = cc3 + cc4;
u = {tmp_u5, tmp_u6};
```

以上 16*16 乘法原理为：

```
                                        AAAAAAAA AAAAAAAA
                                  x     AAAAAAAA AAAAAAAA
                                  ----------------------
                                        AAAAAAAA AAAAAAAA
                                AAAAAAAA AAAAAAAA
                                AAAAAAAA AAAAAAAA
                        +       AAAAAAAA AAAAAAAA
                          ------------------------------------------------------------
                        AAAAAAAA AAAAAAAA AAAAAAAA AAAAAAAA
```

最后，三个功能的运算结果 p，s，u 由功能选择 op 控制输出。

## 四、功能模块设计

功能一 Multiplication 模块，模块文件为 Multiplication.v。此模块将输入的乘法运算结果与 Mout 相加，并判断是否有溢出，如果上溢输出 64'h7FFF_FFFF_FFFF_FFFF，如果下溢输出 64'h8000_0000_0000_0000，否则输出 Mout。

功能二 pixel_accumulation 模块，模块文件为 pixel_accumulatio.v。此模块按照控制码 ww 来选择 Sum 具体加在 Mout 对应的哪些位上，同时判断溢出。

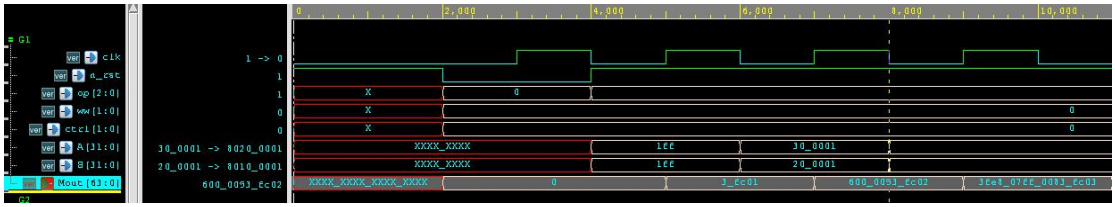
功能三 Plural_operation 模块，模块文件为 Plural_operation.v，此模块完成 16 位复数条件控制乘累加减功能，并对结果进行溢出判断。

## 五、仿真

本设计实验环境为 Ubuntu16.04，Verilog 代码编译软件为 VCS G-2012.09，仿真软件为 Verdi3_I-2014.03-SP2。功能仿真文件为 Testbench.v，功能一中我们输入 A、B 分别为：

```
oper_a = 32'h0000_01ff;
oper_b = 32'h0000_01ff;        //expect: res=3fc01
#2
oper_a = 32'h0030_0001;
oper_b = 32'h0020_0001;        //expect: res=600_0053_fc02
#2
oper_a = 32'h8020_0001;
oper_b = 32'h8010_0001;      //expect: res=3fe8_07ff_0083_fc03
```

仿真结果见图四，可以看到仿真结果与预期结果一致：



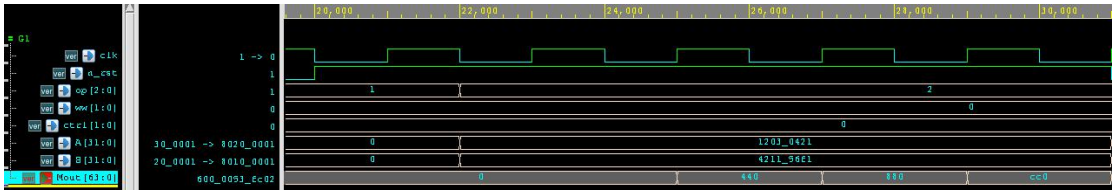
**图四：功能一仿真波形图**

功能二中我们输入 A、B 分别为：

```
a1=8'h12;
a2=8'h03;
a3=8'h04;
a4=8'h21;  //A=32’h1203_0421

b1=8'h42;
b2=8'h11;
b3=8'h56;
b4=8'hf1;   //B=32’h4211_56f1
//     expect: res=440
//     expect: res=880
```

仿真结果见图五，可以看到仿真结果与预期结果一致：



**图五：功能一仿真波形图**

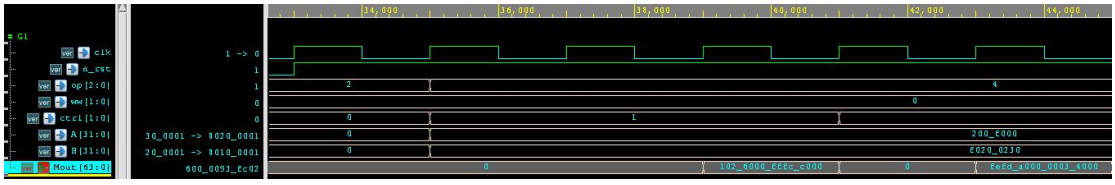
功能三中我们输入 A、B 分别为：

```
t_ctrl=2'b01;
ca1=16'h0200;
ca2=16'hf000;
cb1=16'hf020;
cb2=16'h0230;            //     expect: res=102_6000_fffc_c000

#6
t_ctrl=2'b10;

ca1=16'h0200;
ca2=16'hf000;
cb1=16'hf020;
cb2=16'h0230;            //     expect: res=fefd_a000_0003_4000
```

仿真结果见图六，可以看到仿真结果与预期结果一致：



**图六：功能一仿真波形图**

## 六、结论

本设计实现了多模式乘法器功能，经过仿真检验，结果正确，符合要求。