WARNING – UPDATED SLIDES

- These slides and associated ZIP archives have been updated since they were first posted.
  - Copy your dware dsp_dware/DWDSP_mult_csa.vhd, dsp_test/blend8_rtl.vhd to another location.
  - Remove your old dsp_dware, dsp_test directories, and unzip the updated archives.
- The slides have been updated to reflect the new package hierarchy
- Look at the end of the presentation for the step-by-step procedure – this has been updated as well.

Synthetic Operator Mapping

How is '+' mapped to an implementation?

In *ieee.std_logic_arith*:

```
function "+"(L: UNSIGNED; R: SIGNED) return SIGNED is
        -- pragma label_applies_to plus
        -- synopsys subpgm_id 238
        constant length: INTEGER :=
                 max(L'length + 1, R'length);
    begin
        return plus(CONV_SIGNED(L, length),
            CONV_SIGNED(R, length)); -- pragma label plus
    end;
```

The function 'plus' determines the simulation functionality of '+'

```
 -- both arrays must have range (msb downto 0)
function plus(A, B: SIGNED) return SIGNED is
    variable carry: STD_ULOGIC;
    variable BV, sum: SIGNED (A'left downto 0);
    -- pragma map_to_operator ADD_TC_OP
    -- pragma type_function LEFT_SIGNED_ARG
    -- pragma return_port_name Z
begin
    if (A(A'left) = 'X' or B(B'left) = 'X') then
        sum := (others => 'X');
        return(sum);
    end if;
    carry := '0';
    BV := B;
    for i in 0 to A'left loop
        sum(i) := A(i) xor BV(i) xor carry;
        carry := (A(i) and BV(i)) or
                (A(i) and carry) or
                (carry and BV(i));
    end loop;
    return sum;
end;
```

plus function only defines functionality. ADD_TC_OP defined in synthetic library which contains implementations.

Synthetic Library -- *DWSL.sl*

A Synthetic library has a *.sl* extension (*.sldb* is the compiled form. The file *DWSL.sl* was the synthetic library that was used in Synopsys tutorial. Synthetic libraries contain *operators* and *modules*.

```
library (DWSL.sldb) {

operator (ADD_TC_OP) {
  pin (A) {
    direction : input;
  }
  pin (B) {
    direction : input;
  }
  pin (Z) {
    direction : output;
  }
 }
```

Referenced in Synopsys pragma in 'plus' function.

Operators are defined for various arithmetic functions. The port names do not match the port name implementations. *Modules* define how *operators* map to hardware.

```
module (DWDSP_add) {
   design_library : "DWSL";
   parameter(width) {
     formula : "width('A')";
     hdl_parameter : TRUE ;
  }
  pin (A) {
   direction : input ;  bit_width : "width" ;
   }
  pin (B) {
   direction : input ;  bit_width : "width" ;
   }
  pin (CI) {
  direction : input ; bit_width : "1" ;
   }
  pin (SUM) {
   direction : output; bit_width : "width" ;
  }

  pin (CO) {
  direction : output ; bit_width : "1" ;
  }
pin (OV) {
  direction : output ; bit_width : "1" ;
   }
```

A *module* corresponds to a hardware implementation.

Port definitions match VHDL port definitions.

```
binding (b0) {
   bound_operator : "ADD_TC_OP" ;
   pin_association(A) { oper_pin : A ; }
   pin_association(B) { oper_pin : B ; }
   pin_association(CI) { value : "0" ; }
   pin_association(SUM) { oper_pin : Z ; }
   }

  implementation (rpl) {
    technology : gcmos_unit.db;
   }
  implementation (cla) {
    technology : gcmos_unit.db;
   }
  implementation (csel) {
    technology : gcmos_unit.db;
   }

 }
```

A *binding* determines how *module* pins map to *operator* pins.

Each *implementation* refers to a specific RTL description of this *module*. Can have multiple implementations.

Fixed Point Numbers

- The binary integer arithmetic you are used to is known by the more general term of Fixed Point arithmetic.
  - *Fixed Point* means that we view the decimal point being in the same place for all numbers involved in the calculation.
  - For integer interpretation, the decimal point is all the way to the right

```
  $C0       192.
+ $25     +  37.
--------  --------
  $E5       229.
```

Unsigned integers, decimal point to the right.

A common notation for fixed point is 'X.Y', where X is the number of digits to the left of the decimal point, Y is the number of digits to the right of the decimal point.

Fixed Point (cont).

- The decimal point can actually be located anywhere in the number -- to the right, somewhere in the middle, to the right

Addition of two 8 bit numbers; different interpretations of results based on location of decimal point

```
 $11          17          4.25          0.07
+ $1F       +  31       +  7.75       +  0.12
--------    --------    --------      --------
 $30          48         12.00          0.19
```

xxxxxxxx.0 — decimal point to right. This is 8.0 notation.

xxxxxx.yy — two binary fractional digits. This is 6.2 notation.

0.yyyyyyyy — decimal point to left (all fractional digits). This is 0.8 notation.

Unsiged Overflow

- Recall that a carry out of the Most Significant Digit is an unsigned overflow. This indicates an error - the result is NOT correct!

Addition of two 8 bit numbers; different interpretations of results based on location of decimal point

```
 $FF        255          63.75          0.99600
+ $01     +    1       +  0.25        +  0.00391
--------  --------     -----------    -----------
 $00          0           0               0
```

xxxxxxxx.0 — decimal point to right

xxxxxx.yy — two binary fractional digits (6.2 notation)

0.yyyyyyyy — decimal point to left (all fractional digits). This 0.8 notation

Saturating Arithmetic

- Saturating arithmetic means that if an overflow occurs, the number is clamped to the maximum possible value.
  - Gives a result that is closer to the correct value
  - Used in DSP, Graphic applications.
  - Requires extra hardware to be added to binary adder.
  - Pentium MMX instructions have option for saturating arithmetic.

```
 $FF        255          63.75          0.99600
+ $01     +    1       +  0.25        +  0.00391
--------  --------     -----------    -----------
 $FF        255          63.75          0.99600
```

xxxxxxxx.0 — decimal point to right

xxxxxx.yy — two binary fractional digits.

0.yyyyyyyy — decimal point to left (all fractional digits)

Saturating Arithmetic

The MMX instructions perform SIMD operations between MMX registers on packed bytes, words, or dwords.

The arithmetic operations can made to operate in Saturation mode.

What saturation mode does is clip numbers to Maximum positive or maximum negative values during arithmetic.

In normal mode: FFh + 01h = 00h (unsigned overflow)
In saturated, unsigned mode: FFh + 01 = FFh (saturated to maximum value, closer to actual arithmetic value)

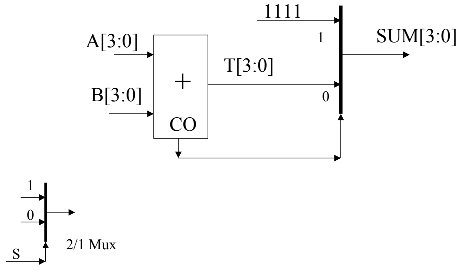
In normal mode: 7fh + 01h = 80h (signed overflow)

In saturated, signed mode: 7fh + 01 = 7fh (saturated to max value)

Saturating Adder: Unsigned and 2'Complement

- For an unsigned saturating adder, 8 bit:
  - Perform binary addition
  - If Carryout of MSB =1, then result should be a $FF.
  - If Carryout of MSB =0, then result is binary addition result.
- For a 2's complement saturating adder, 8 bit:
  - Perform binary addition
  - If Overflow = 1, then:
    - If one of the operands is negative, then result is $80
    - If one of the operands is positive, then result is $7f
  - If Overflow = 0, then result is binary addition result.

Saturating Adder: Unsigned, 4 Bit example



dwdsp_arith.vhd

Create a package that does saturating arithmetic, plus other functions for DSP. Create a Synthetic Library that maps the operators.

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package dwdsp_arith is

   function dspmult(A: UNSIGNED; B: UNSIGNED) return
UNSIGNED;
   function "+"(L: UNSIGNED; R: UNSIGNED) return
UNSIGNED;

end dwdsp_arith;
```

'+' does unsigned saturating add. Will talk about *dspmult* later.

unsigned_plus function in d*wdsp_arith.vhd*

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
function unsigned_plus(A, B: UNSIGNED) return UNSIGNED is
        variable carry: STD_ULOGIC;
        variable BV, sum: UNSIGNED (A'left downto 0);
        -- pragma map_to_operator ADD_UNS_OP
        -- pragma type_function LEFT_UNSIGNED_ARG
        -- pragma return_port_name Z
    begin
        if (A(A'left) = 'X' or B(B'left) = 'X') then
            sum := (others => 'X');
            return(sum);
        end if;
        carry := '0';  BV := B;
        for i in 0 to A'left loop
            sum(i) := A(i) xor BV(i) xor carry;
            carry := (A(i) and BV(i)) or
                    (A(i) and carry) or
                    (carry and BV(i));
        end loop;
       if (carry = '1') then    -- saturate
         sum := (others => '1');
       end if;
        return sum;
    end;
```

Pragmas for Synthetic operator definition.

Functionality for addition

Saturate done here.

'+' function in d*wdsp_arith.vhd*

```
function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED is
        -- pragma label_applies_to plus
      constant length: INTEGER := max(L'length, R'length);
    begin
        return unsigned_plus(CONV_UNSIGNED(L, length),
             CONV_UNSIGNED(R, length)); -- pragma label plus
    end;
```

Just a wrapper around the *unsigned_plus* function – maps '+' to *unsigned_plus*.

DWDSP_add.vhd - module for saturating addition

```
library ieee;
use ieee.std_logic_1164.all;

entity DWDSP_add is
  generic(width : POSITIVE);
  port(A,B : std_logic_vector(width-1 downto 0);
       CI : std_logic;
       SUM : out std_logic_vector(width-1 downto 0)   );
end DWDSP_add;
```

Will create designware library called DWDSP and place modules in this library.

The synthetic library will be called *DWDSP.sl*.

DWDSP_add_cla.vhd - CLA architecture for saturating addition

```
library IEEE, DW01,synopsys;
use IEEE.std_logic_1164.all;
use DW01.DW01_components.all;
use synopsys.attributes.all;
architecture cla of DWDSP_add is
  attribute implementation: STRING;
  attribute implementation of U1 : label is "cla";
  signal tsum: std_logic_vector(width-1 downto 0);
  signal satval : std_logic_vector(width-1 downto 0);
  signal tco: std_logic;

begin
  U1: DW01_add generic map (width => width)
       port map (CI =>CI, A =>a, B=>b,  SUM => tsum, CO => tco);
  satval <= (others => '1');
  sum <= tsum when (tco = '0') else satval;
end cla;
```

Note that DW01 adder is used with extra logic mux logic for saturation. Explicit control of CLA implementation via attributes.

Can use modules from other DW libraries!

Example Fixed Point Application

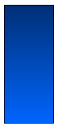
Colors in Computer Graphics applications represented by Red, Green, Blue (RGB) components.

Each component (RGB) is 8 bits; hence the term 24 bit color.

As an 0.8 Fixed point number, colors range from:

0.0 =< color < 1.0

(dark colors) (light colors)



0.0 = % 00000000

Blue variation

0.99 = % 11111111

Blend Operation

A blend operation takes two colors and blends them together to form a new color. The *Blend Factor* (F) controls how much each color contributes

Cnew = Ca * F + (1 - F) Cb

If F is 0.5 then the new color is an equal blend of Ca, Cb.

If F is 0.0, then new color is simply Cb.

If F is 1.0, then new color is simply Ca.

Representing 1.0

When the multiplication Ca * F is performed, if F = 1.0 want the result to be exactly equal to the original value 'Ca'.

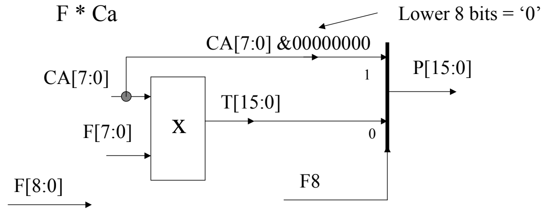
However, the closest we can get to 1.0 using 8 bits (assuming 0.8 fixed point notation) is $0.11111111_2 = 0.996_{10}$

0.996 x Ca is NOT EQUAL to Ca!

To solve this problem, we will use 9 bits to represent the 'F' value. The lower 8 bits will be the fractional representation of F. If F=1.0, then the MSB of F is equal to a '1', and the other bits are a don't care.

When multiplying Ca * F, will use the lower 8 bits of F for the multiply. If the MSB of F = '1', then ignore output of multiplier and use 'Ca'.

F * Ca



If F = 1.0, then F = '1xxxxxxxx' (MSB of F = '1').

Note that when F = 1.0, the 16 bit result is the original CA value with the least significant 8 bits padded with '0's. In a datapath, cannot keep expanding data width, so will typically drop the 8 least signficant bits of the product.

dspmult operation in *dwdsp_arith.vhd*

```
-- very restrictive '*'
-- always assume that B'width = A'width +1
-- also assume that vectors are declared with 'downto' range
function dspmult(A: UNSIGNED; B: UNSIGNED) return UNSIGNED is
      -- pragma map_to_operator MULT_UNS_OP
      -- pragma type_function MULT_UNSIGNED_ARG
      -- pragma return_port_name Z
        variable result  : unsigned ((2*A'length)-1 downto 0);
    begin
     if (B(B'high) = '1') then
       result((2*A'length)-1 downto A'length) := A;
       for i in 0 to A'length-1 loop
         result(i) := '0';
       end loop;
     else
        result  := normal_mult(unsigned(A),unsigned(B((B'high)-1
downto B'low)) );
     end if;
     return   result;
    end;
```

Pragmas for Synthetic operator definition.

Multiplication by 1.0 (B can represent 1.0)

normal_mult defined elsewhere in *dwdsp_arith* – does the NxN bit multiply if B is not equal to 1.0

dwdsp_mult.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;


entity DWDSP_mult is

  generic(width : POSITIVE);
  port(A : std_logic_vector(width-1 downto 0);
       F : std_logic_vector(width downto 0);
       P : out std_logic_vector((2*width)-1 downto 0)   );
end DWDSP_mult;
```

Module for implementing A*F .

Note that F is the 2nd operand, so it will be the *righthand* operand in '*L * R*'.

Note that F is always 1 bit wider than 'A', returned product is 2 * width of A.

dwdsp_mult_csa.vhd

You will need to fill this out. Use the multiplier from the *DW02* designware library, and use the 'csa' architecture (CSA = carry save array). See the DWSL_add_cla.vhd file for structure hints.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DW02_mult is
   generic( A_width: NATURAL;           -- multiplier wordlength
            B_width: NATURAL);          -- multiplicand wordlength
   port(A : in std_logic_vector(A_width-1 downto 0);
        B : in std_logic_vector(B_width-1 downto 0);
        TC : in std_logic;  -- signed -> '1', unsigned -> '0'

        PRODUCT : out
              std_logic_vector(A_width+B_width-1 downto 0));

end DW02_mult;
```

Set TC='0' since we want an unsigned multiplier.

DWDSP.sl Synthetic Library

I have already created the DWDSP.sl Synthetic library for you. You do not need to modify this file.

It contains two operators – '+' and '*'.

The '+' is mapped to the *DWDSP_add.vhd* module that implements saturating addtion.

The '*' is mapped to the *DWDSP_mult.vhd* module – you will need to complete the *DWDSP_mult_cla.vhd* architecture.

dwdsp_arith_unsigned.vhd

This package provides a wrapper around '+', and maps the '*' to the *dspmult* function from *dwdsp_arith*. Both functions accept *std_logic_vector* values and convert them to the *unsigned* type.

This package also defines the *oneminus* function (discussed later)

```
library ieee,dwdsp, synopsys;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use synopsys.attributes.all;
use dwdsp.dwdsp_arith.all;

package dwdsp_arith_unsigned is
   function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
return STD_LOGIC_VECTOR;
   function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
return STD_LOGIC_VECTOR;
   function oneminus(L: STD_LOGIC_VECTOR) return
STD_LOGIC_VECTOR;

end dwdsp_arith_unsigned;
```

'+' in *dwdsp_arith_unsigned.vhd*

```
function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
STD_LOGIC_VECTOR is
        constant length: INTEGER := maximum(L'length, R'length);
        variable result  : STD_LOGIC_VECTOR (length-1 downto 0);

   constant r0 : resource := 0;
   attribute map_to_module of r0: constant is "DWDSP_add";
   attribute implementation of r0: constant is "cla";
   attribute ops of r0: constant is "a0";
    begin
        result  := UNSIGNED(L) + UNSIGNED(R); -- pragma label a0

       return   std_logic_vector(result);
    end;
```

This function converts std_logic_vector operands to unsigned, then uses the '+' operator from *dsp_arith*.

It also forces the mapping of '+' to the *DWDSP_add* module with architecture 'cla'.

Mapping '+' to DWDSP_add

- Forcing the mapping of '+' of DWDSP_add, architecture 'cla' within *dsp_arith_unsigned* is sub-optimal
  - Ideally, would like to do this from a dc_shell script
  - The default is to map '+', '*' to the operators defined in the standard synthetic library (standard.sldb), which maps these to DW01_add, DW02_mult respectively.
- Unfortuntely, I have been unable to figure out the correct magic to add to the *dc_shell* scripts to force use of *DWDSP_add*, *DWDSP_mult*
  - *dc_shell* stubbornly selects normal DW01_add, DW02_mult mappings no matter what I try.
  - We will live with this solution for now, but there is probably a better way.
- Your RTL and behavioral code should use the *dwdsp_arith_unsigned* package, and use the '+', '*' operators for addition, multiplication.

'*' in *dwdsp_arith_unsigned.vhd*

```
function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return
STD_LOGIC_VECTOR is
   variable result  : unsigned ((2*L'length)-1 downto 0);
   variable a : unsigned((L'length-1) downto 0);
   variable b : unsigned((R'length-1) downto 0);
   constant r1 : resource := 0;
   attribute map_to_module of r1: constant is "DWDSP_mult";
   attribute implementation of r1: constant is "csa";
   attribute ops of r1: constant is "a1";
   begin
       a := unsigned(L);
       b := unsigned(R);
        result  := dspmult(a,b); -- pragma label a1
        return  std_logic_vector(result);
   end;
```

dspmult function from *dsp_arith* package.

What about "1 – F" ?

For speed purposes, will represent 1-F as simply the one's complement of 'F'. This will be inaccurate by 1 LSB, but is fast.

Need to consider cases of 1.0, 0.0, and default.

-- F = 1.0
if (F(msb) = '1') then F_minus = 0;

Else If (F = 0) then F_minus = "10000…0";

Else F_minus(msb) = 0
F_minus(msb-1 downto 0) = F (msb-1 downto 0);

Do not need to define a synthetic module for this because of the simplicity of the operation – normal logic synthesis via a VHDL function will be sufficient.

1-F function (*oneminus* in *dwdsp_arith_unsigned.vhd*)

```
-- does 1-L function where msb of L stands for 1.0
-- minus is simply one's complement


   function oneminus(L: STD_LOGIC_VECTOR)
            return STD_LOGIC_VECTOR is
     variable result  : STD_LOGIC_VECTOR (L'length-1 downto 0);
     variable zero :STD_LOGIC_VECTOR (L'length-1 downto 0);
     begin
       zero := (others => '0');
       if (L(L'high) = '1') then
         result := (others => '0');
       elsif (L = zero) then
         result := (others => '0');
         result(L'length-1) := '1';
       else
         result := not(L);
         result(L'length-1) := L(L'high);
       end if;
       return result;
     end;
```

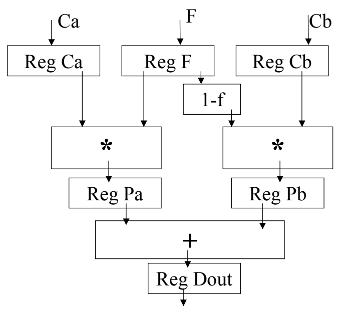
F = 1.0, return 0

F = 0.0, return 1.0

Else return 1-F (one's complement)

blend8 Datapath

To test your *DWDSP_mult_cla*, create an 8 bit blend datapath as shown below:



All datapaths are 8 bits including multiplier output (only keep 8 most significant bits).

The '+', '*' are as defined in *dwdsp_arith_unsigned*

Fully pipelined datapath, all registers loaded every clock.

blend8.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

entity blend8 is
   port ( clk,reset: std_logic;
          ca,cb: in std_logic_vector(7 downto 0);
          f: in std_logic_vector(8 downto 0);
          p: out std_logic_vector(7 downto 0) );
end blend8;
```

Reset is a synchronous reset – all registers zero when reset = '1'.

blend8_rtl.vhd

Use library *dwdsp*, package *dwdsp_arith_unsigned*.

```
library ieee,dwdsp;
use ieee.std_logic_1164.all;
use dwdsp.dwdsp_arith_unsigned.all;


architecture rtl of blend8 is

 --- empty architecture – fill this out

 end rtl;
```

dsp_test.zip Archive

- Unpacks four VHDL source directories. Install these under vhdl_course/src. Makefiles for each are in their respective directories.
  - synopsys/ -- install as Modelsim library, compile this first. This is used by 'dwdsp_arith_unsigned' package.
  - dwdsp/ -- install as Modelsim library, compile this second.
  - gcmos/ -- update to gcmos library (fixed a problem with 'dfr') compile this third.
  - dsp_test/ -- install as Modelsim library. Provides a testbench for your blend implementation.
- Once *synopsys*, *dwdsp*, *gcmos* libraries are compiled, will not need to compile these again.

dsp_test Library

Files are:
dsp_test/Makefile.dsp_test --- makefile
dsp_test/blend8.vhd --- blend8 entity
dsp_test/blend8_rtl.vhd -- empty architecture – fill this out, use for synthesis
dsp_test/blend8_gate.vhd -- empty architecture, replace this with synthesized gate level architecture
dsp_test/dsp_tbblend.vdh --- test bench, contains configurations for rtl, gate architectures

dsp_test/tbblend_gold.log -- log file of golden simulation (gate and rtl architecture simulations should match)

dsp_dware.zip Archive

Will expand to dsp_dware directory – should be placed under vhdl_course/synopsys. This is the directory that should be used for Synopsys synthesis. Important files are (not all listed):

DWDSP_mult_csa.vhd – architecture for '*' implementation

compile_dwdsp_lib.script – dc_shell script for compiling DWDSP modules, use this after any changes to DWDSP* files.

rtl/blend8.vhd -- place both blend8 entity and RTL architecture in here for synthesis.

blend8.script – dc_shell script for synthesizing 'rtl/blend8.vhd' using the DWDSP synthetic library. Output file will be 'gate/blend8_gate.vhd'.

Steps to Complete this Assignment

- Complete the *dsp_test/blend8_rtl.vhd* architecture to implement the blend8 datapath and match the golden output file
  - Uses '*', '+', *oneminus* functions from *dsp_arith_unsigned* package.
- Edit the *dsp_dware/rtl/blend8.vhd file* and place the 'rtl' architecture from *dsp_test/blend8_rtl.vhd* in here.
- Complete the *DWDSP_mult_csa.vhd* file to implement the '*' function as discussed
  - Use '*dc_shell –f compile_dwdsp_lib.script* ' to compile
- Synthesize a gate level architecture
  - Use 'dc_shell –f blend8.vhd' – will produce gate/blend8_gate.vhd
- Copy 'gate/blend8_gate.vhd' to dsp_test/blend8_gate.vhd, compile in Modelsim, and see if results of gate level configuration simulation matches the RTL simulation.

blend8.rpt File

After synthesizing your design using *dc_shell* and the *blend8.script* file, look inside the *blend8.rpt* file.

The implementation section should show *dwdsp_mult, dwdsp_add* operators being used.

```
Implementation Report

===============================================================================
|                  |                  | Current          | Set              |
| Cell             | Module           | Implementation   | Implementation   |
===============================================================================
| add_47/a0/plus   | DWDSP_add        | cla              | cla              |
| mul_44/a1        | DWDSP_mult       | csa              | csa              |
| mul_45/a1        | DWDSP_mult       | csa              | csa              |
===============================================================================
```