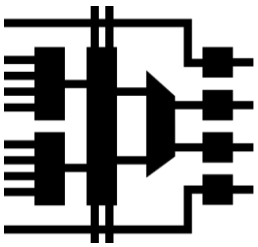

FPGA-SYSTEMS.RU

сообщество FPGA разработчиков

ПОДГОТОВИЛ: Amurak

Реализация базовых компонентов ЦОС

Комплексный умножитель

2021

Оглавление

FPGA SYSTEMS
сообщество FPGA разработчиков

Контакты автора

Лотник Виталий

e-mail:

lve@fpga-systems.ru

Привет!

Представленный материал является частью цикла статей, в которых рассматривается RTL реализация базовых компонентов цифровой обработки сигналов. В качестве языка описания используется VHDL-2008, проектирование ведется с учетом особенностей ПЛИС фирмы Xilinx.

Данная статья подготовлена для информационно-образовательного портала FPGA / ПЛИС разработчиков FPGA-Systems.ru

Исходные коды из статьи выложены на github

Приятного прочтения.

Контакты комьюнити

сайт:

fpga-systems.ru

e-mail:

admin@fpga-systems.ru

Дата	Версия	Автор	Изменения
24.05.2021	1.0	Amurak	Начальный релиз документа

***О найденных опечатках и замечаниях просим сообщить admin@fpga-systems.ru**

Аннотация

В данной статье рассматриваются особенности реализации одного из базовых компонентов цифровой обработки сигналов - комплексного умножителя. Разобраны способы разбиения умножения комплексных чисел на простейшие арифметические операции. Представлено RTL описание параметризируемого модуля на языке VHDL-2008.

1. Теоретическая часть

Комплексное умножение широко применяется при обработке комплексных сигналов. Оно встречается, например, при реализации таких блоков, как: цифровые повышающие/понижающие преобразователи, комплексные фильтры и корреляторы, блоки быстрого преобразования Фурье, схемы демодуляции и т.д.

Умножение двух комплексных чисел определяется как:

$$(A + ja) \times (B + jb) = C + jc$$

Если раскрыть скобки, то мы получим следующее выражение:

$$(A + ja) \times (B + jb) = (A \times B - a \times b) + j(A \times b + a \times B)$$

При этом:

$$C = A \times B - a \times b;\ c = A \times b + a \times B$$

Схематическое изображение данных операций представлено на рисунке 1.

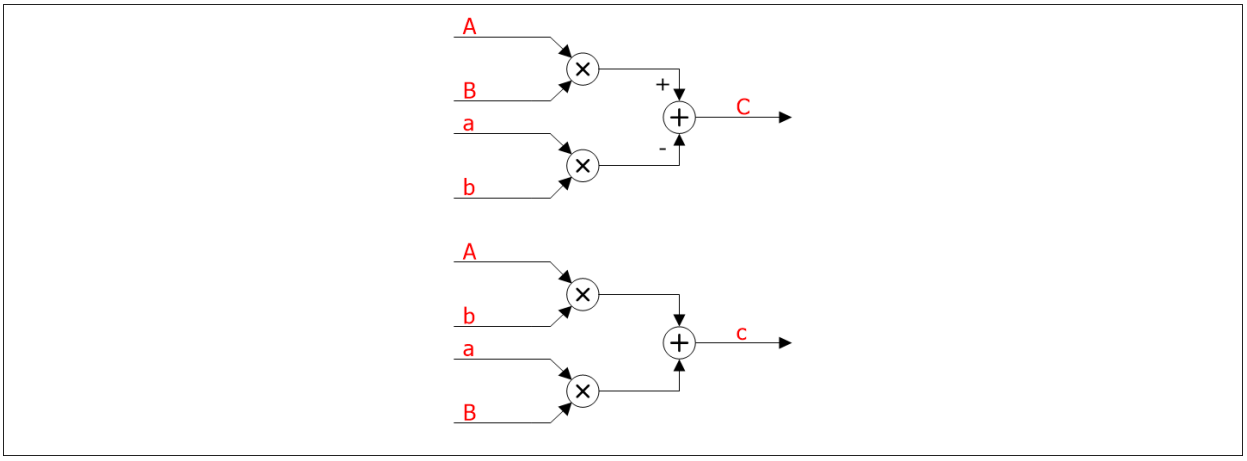

Рисунок 1 – Схема комплексного умножения

Для реализации данной схемы требуется четыре умножителя и два сумматора. Обозначим данную схему как «Схема 1».

Возможен и другой способ получения тех же значений:

$$C = A \times (B + b) - (a + A) \times b = A \times B + A \times b - a \times b - A \times b = A \times B - a \times b$$

$$c = A \times (B + b) + (a - A) \times B = A \times B + A \times b + a \times B - A \times B = A \times b + a \times B$$

Схематическое изображение выделенных операций представлено на рисунке 2.

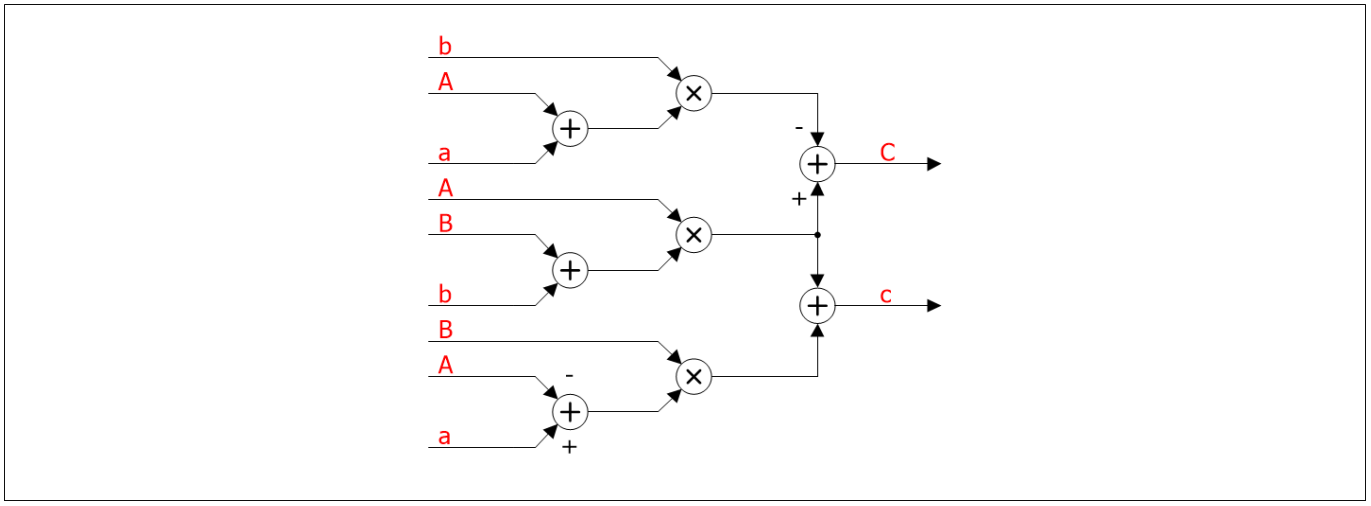

Рисунок 2 – Альтернативная схема комплексного умножения

Для реализации данной схемы требуется три умножителя и пять сумматоров. Обозначим данную схему как «Схема 2».

2. Функциональные схемы

Для достижения максимально возможной частоты обработки при реализации комплексного умножения на ПЛИС фирмы Xilinx следует использовать аппаратный блок DSP48, который содержит в себе умножитель, сумматор, предварительный сумматор, а так же наборы триггеров для входных/выходных данных и промежуточных результатов вычислений. В качестве примера будут рассмотрены схемы на основе DSP48E1.

Один из таких вариантов реализации «Схемы 1» представлен на рисунке 3.

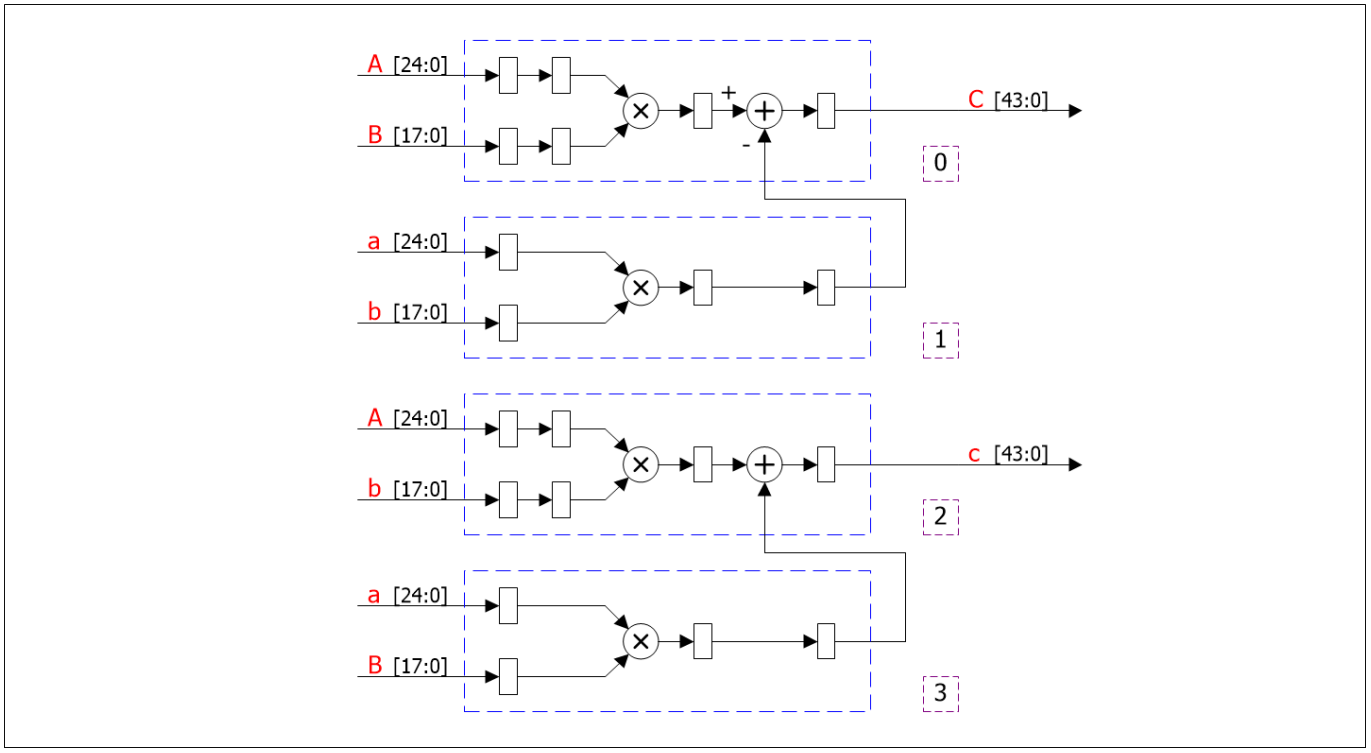

Рисунок 3 – реализация «Схемы 1» на блоках DSP48

Как видно из рисунка, данный вариант комплексного умножителя полностью укладывается в ресурсы аппаратных блоков DSP48E1 без использования дополнительной логики. Блоки 0 и 1 используются для расчета действительной части, 2 и 3 – мнимой части выходного числа.

Ввиду того, что умножитель DSP48E1 имеет разрядность 25x18, это накладывает ограничения на максимальную разрядность входных данных. В данном случае для входа A выбрана разрядность 25 бит, для входа B – 18 бит. Выход в таком случае будет иметь разрядность 44 бита.

Латентность данной схемы равняется четырем тактам.

«Схему 2» можно реализовать аналогичным образом, если использовать предварительный сумматор, как показано на рисунке 4.

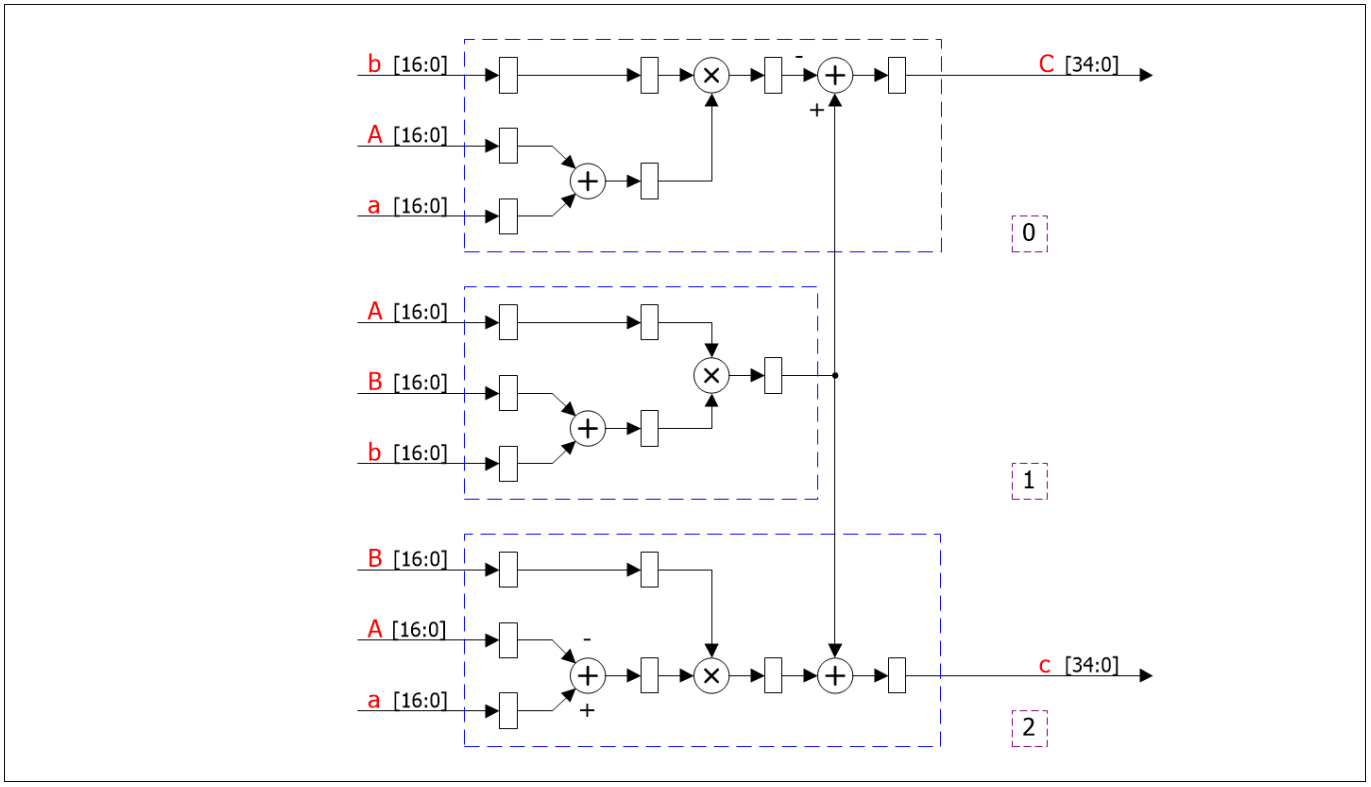

Рисунок 4 - реализация «Схемы 2» на блоках DSP48

Как видно из рисунка, данный вариант также полностью укладывается в ресурсы DSP48E1 и при этом используется на один аппаратный блок меньше. Блоки 0 и 1 используются для расчета действительной части, 1 и 2 – мнимой части выходного числа.

В данной схеме дополнительные ограничения на разрядность входных данных вносит тот факт, что данные поступают на предварительный сумматор и умножитель в различных комбинациях, поэтому максимальная разрядность входов A и B равна 17 битам. Выход в таком случае будет иметь разрядность 35 бит.

Латентность данной схемы также равняется четырем тактам.

3. RTL описание

Для описания комплексных чисел будет использоваться пользовательский тип t_iq , который объявлен в пакете pkg_rtl_modem_types и включает в себя два поля: i и q типа signed. VHDL 2008 позволяет объявлять данные поля без ограничений.

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

package pkg_rtl_modem_types is
    type t_iq is record
        i : signed;
        q : signed;
    end record;
end;
```

Листинг 1 – Объявление типа t_iq

Объявим интерфейс комплексного умножителя:

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

library rtl_modem;
    use rtl_modem.pkg_rtl_modem_types.all;

entity complex_multiplier is
    generic(
          g_a_dw                    : integer := 12
        ; g_b_dw                    : integer := 12
        ; g_type                    : integer := 1
    );
    port(
          iCLK                      : in std_logic
        ; iV                        : in std_logic
        ; iA                        : in t_iq
        ; iB                        : in t_iq
        ; oV                        : out std_logic
        ; oC                        : out t_iq
    );
end;
```

Листинг 2 – Интерфейс комплексного умножителя

Порт iCLK – сигнал тактирования модуля, iV – сигнал валидности входных данных, iA и iB – входные комплексные числа, oV – валидность выходных данных, oC – результат умножения.

Параметры g_a_dw и g_b_dw задают разрядность соответствующих входов A и B. Параметр g_type определяет, какая из схем умножения будет реализована: 1 – «схема 1», 2 – «схема 2».

Объявим сигналы для приема данных с входных и выдачи данных на выходные порты:

```
architecture behavioral of complex_multiplier is
    constant c_c_dw : integer := g_a_dw + g_b_dw + 1;

    -- input buffers
    signal ib_v : std_logic;
    signal ib_a : t_iq(i(g_a_dw - 1 downto 0), q(g_a_dw - 1 downto 0));
    signal ib_b : t_iq(i(g_b_dw - 1 downto 0), q(g_b_dw - 1 downto 0));

    signal pipe_v : std_logic_vector(0 to 4) := (others => '0');

    -- output buffers
    signal ob_v : std_logic;
    signal ob_c : t_iq(i(c_c_dw - 1 downto 0), q(c_c_dw - 1 downto 0;

begin
-------------------------------------------------------------------------------
-- input
-------------------------------------------------------------------------------
    ib_v                            <= iV;
    ib_a                            <= iA;
    ib_b                            <= iB;

    pipe_v(0)                       <= ib_v;
    pipe_v(1 to pipe_v'high)        <= pipe_v(0 to pipe_v'high - 1)
                                                    when rising_edge(iCLK);
-------------------------------------------------------------------------------
-- output
-------------------------------------------------------------------------------
    oV                              <= ob_v;
    oC                              <= ob_c;
end;
```

Листинг 3 – Описание основных сигналов

Сигнал pipe_v предназначен для задержки сигнала валидности со входа на выход, длина конвейера при этом определяется латентностью схемы (напомним, у обеих реализация умножителя она равна четырем).

Разрядность выходных данных определяется операциями умножения и сложения. Тело основного процесса, реализующего умножитель по «схеме 1» имеет вид:

```
    p_main : process(iCLK)
    begin
        if rising_edge(iCLK) then
            if pipe_v(0) = '1' then
                dsp0_areg1 <= ib_a.i;                       -- A
                dsp1_areg1 <= ib_a.q;                       -- a
                dsp2_areg1 <= ib_a.i;                       -- A
                dsp3_areg1 <= ib_a.q;                       -- a
                dsp0_breg1 <= ib_b.i;                       -- B
                dsp1_breg1 <= ib_b.q;                       -- b
                dsp2_breg1 <= ib_b.q;                       -- b
                dsp3_breg1 <= ib_b.i;                       -- B
            end if;

            if pipe_v(1) = '1' then
                dsp0_areg2 <= dsp0_areg1;                   -- A
                dsp0_breg2 <= dsp0_breg1;                   -- B
                dsp1_mreg  <= dsp1_areg1 * dsp1_breg1;      -- a x b
                dsp2_areg2 <= dsp2_areg1;                   -- A
                dsp2_breg2 <= dsp2_breg1;                   -- b
                dsp3_mreg  <= dsp3_areg1 * dsp3_breg1;      -- a x B
            end if;

            if pipe_v(2) = '1' then
                dsp0_mreg <= dsp0_areg2 * dsp0_breg2;       -- A x B
                dsp1_preg <= resize(dsp1_mreg, c_c_dw);     -- a x b
                dsp2_mreg <= dsp2_areg2 * dsp2_breg2;       -- A x b
                dsp3_preg <= resize(dsp3_mreg, c_c_dw);     -- a x B
            end if;

            if pipe_v(3) = '1' then
                dsp0_preg <=                                -- A x B - a x b
                    resize(dsp0_mreg, c_c_dw) -
                    dsp1_preg;
                dsp2_preg <=                                -- A x b + a x B
                    resize(dsp2_mreg, c_c_dw) +
                    dsp3_preg;
            end if;
        end if;
    end process;
```

Листинг 4 – Исходный код основного процесса «Схемы 1»

По сути, в теле описаны преобразования всех триггеров, представленных на рисунке 3. Аналогичным образом описывается «схема 2»:

```
    p_main : process(iCLK)
    begin
        if rising_edge(iCLK) then
            if pipe_v(0) = '1' then
                dsp0_breg1 <= ib_b.q;                       -- b
                dsp0_areg1 <= ib_a.i;                       -- A
                dsp0_dreg  <= ib_a.q;                       -- a
                dsp1_breg1 <= ib_a.i;                       -- A
                dsp1_areg1 <= ib_b.i;                       -- B
                dsp1_dreg  <= ib_b.q;                       -- b
                dsp2_breg1 <= ib_b.i;                       -- B
                dsp2_areg1 <= ib_a.i;                       -- A
                dsp2_dreg  <= ib_a.q;                       -- a
            end if;

            if pipe_v(1) = '1' then
                dsp0_breg2 <= dsp0_breg1;                   -- b
                dsp0_adreg <=                               -- A + a
                    resize(dsp0_areg1, c_ad_a_dw) +
                    resize(dsp0_dreg,  c_ad_a_dw);
                dsp1_breg2 <= dsp1_breg1;                   -- A
                dsp1_adreg <=                               -- B + b
                    resize(dsp1_areg1, c_ad_b_dw) +
                    resize(dsp1_dreg,  c_ad_b_dw);
                dsp2_breg2 <= dsp2_breg1;                   -- B
                dsp2_adreg <=                               -- a - A
                    resize(dsp2_dreg,  c_ad_a_dw) -
                    resize(dsp2_areg1, c_ad_a_dw);
            end if;

            if pipe_v(2) = '1' then
                dsp0_mreg <= dsp0_breg2 * dsp0_adreg;       -- b x (A + a)
                dsp1_mreg <= dsp1_breg2 * dsp1_adreg;       -- A x (B + b)
                dsp2_mreg <= dsp2_breg2 * dsp2_adreg;       -- B x (a - A)
            end if;

            if pipe_v(3) = '1' then
                dsp0_preg <=                                -- A x (B + b) - b x (A + a)
                    resize(dsp1_mreg, c_c_dw) -
                    resize(dsp0_mreg, c_c_dw);
                dsp2_preg <=                                -- A x (B + b) + B x (a - A)
                    resize(dsp1_mreg, c_c_dw) +
                    resize(dsp2_mreg, c_c_dw);
            end if;
        end if;
    end process;
```

Листинг 5 – Исходный код основного процесса «Схемы 2»

4. Синтез

Для синтеза будет использоваться Vivado версии 2020.2.

Vivado не позволяет синтезировать модули, порты которых имеют неограниченные поля, поэтому для синтеза мы напишем компонент – обертку:

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

library rtl_modem;
    use rtl_modem.pkg_rtl_modem_types.all;

entity complex_multiplier_wrap is
    generic(
          g_a_dw                    : integer := 25
        ; g_b_dw                    : integer := 18
        ; g_type                    : integer := 1
    );
    port(
          iCLK                      : in std_logic
        ; iV                        : in std_logic
        ; iA_I                      : in signed(g_a_dw - 1 downto 0)
        ; iA_Q                      : in signed(g_a_dw - 1 downto 0)
        ; iB_I                      : in signed(g_b_dw - 1 downto 0)
        ; iB_Q                      : in signed(g_b_dw - 1 downto 0)
        ; oV                        : out std_logic
        ; oC_I                      : out signed(g_a_dw + g_b_dw downto 0)
        ; oC_Q                      : out signed(g_a_dw + g_b_dw downto 0)
    );
end;
```

Листинг 6 – Обертка комплексного умножителя

Синтезируем наш модуль по «схеме 1» с разрядностями 25 x 18.

В логе синтеза мы увидим следующее сообщение:

```
DSP: Preliminary Mapping    Report (see note below)
+------------------+------------------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|Module Name       | DSP Mapping            | A Size | B Size | C Size | D Size | P Size | AREG | BREG | CREG | DREG | ADREG | MREG | PREG |
+------------------+------------------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|complex_multiplier | (A2*B2)'              | 25     | 18     | -      | -      | 43     | 1    | 1    | -    | -    | -     | 1    | 1    |
|complex_multiplier | (-PCIN+(A''*B'')'+1-1)' | 25     | 18     | -      | -      | 44     | 2    | 2    | -    | -    | -     | 1    | 1    |
|complex_multiplier | (A2*B2)'              | 25     | 18     | -      | -      | 43     | 1    | 1    | -    | -    | -     | 1    | 1    |
|complex_multiplier | (PCIN+(A''*B'')')'    | 25     | 18     | -      | -      | 44     | 2    | 2    | -    | -    | -     | 1    | 1    |
+------------------+------------------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
```

Как и ожидалось, все описанные в исходном коде триггеры были размещены в блоках DSP48. Видно, что для подачи результата умножения с одного аппаратного блока на сумматор другого используется порт PCIN.

Аналогично синтезируем наш модуль по «схеме 2» с разрядностями 17 x 17.

Лог для данного случая:

```
DSP: Preliminary Mapping    Report (see note below)
+------------------+----------------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|Module Name       | DSP Mapping          | A Size | B Size | C Size | D Size | P Size | AREG | BREG | CREG | DREG | ADREG | MREG | PREG |
+------------------+----------------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|complex_multiplier | ((D'+A2)*B'')'      | 17     | 17     | -      | 17     | 35     | 1    | 2    | -    | 1    | 1     | 1    | 0    |
|complex_multiplier | (PCIN-((D'+A2)*B'')')' | 17     | 17     | -      | 17     | 35     | 1    | 2    | -    | 1    | 1     | 1    | 1    |
|complex_multiplier | (C+((D'-A2)*B'')')'  | 17     | 17     | 35     | 17     | 35     | 1    | 2    | 0    | 1    | 1     | 1    | 1    |
+------------------+----------------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
```

В данном случае, поскольку необходимо передать промежуточный результат с одного аппаратного блока на два других, используются порты PCIN и C, что снижает максимально возможную частоту обработки в сравнении со «схемой 1».

По итогу мы выяснили, что «схема 1» обладает преимуществами в разрядности входных данных и позволяет выжать максимум скорости из аппаратных блоков ПЛИС. Однако в проектах, где не требуется такая большая разрядность, «схема 2» позволяет экономить один DSP48 на каждую операцию комплексного умножения. Кроме того, максимальная рабочая частота чаще всего ограничена другими компонентами проекта, поэтому в этом плане разница между «схемами» несущественна.

Ссылки

1. https://github.com/vlotnik/rtl_modem - исходные коды, представленные в статье
2. https://www.xilinx.com/support/documentation/user_guides/ug193.pdf - гайд по использованию аппаратных блоков DSP48
3. https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf - описание аппаратного блока DSP48E1
4. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf - описание аппаратного блока DSP48E2