نیمسال تحصیلی دوم ۱۴۰۰ - ۱۳۹۹

استاد درس: دکتر محمدرضا پورفرد

عنوان پروژه: واحد محاسبه‌گر منطقی (Arithmetic Logic Unit)

اعضای گروه:

نام	نام خانوادگی	شماره دانشجویی
عرفان	راستی	۹۸۲۳۰۳۴
آرمان	عباسی	۹۸۲۳۰۶۲
دانیال	غریبی	۹۸۲۳۰۶۶
سید احمدرضا	موسوی	۹۸۲۳۰۸۶

فهرست

مقدمه

امروزه در بسیاری از کاربردها، از قطعات الکرونیکی برای تحلیل و پردازش داده‌ها استفاده می‌شود. این قطعات که به طور کلی آن‌ها را پردازنده می‌نامیم، به منزله مغز یا فرمانده هر سیستم ایفای نقش می‌کند؛ تا اتفاقی دلخواه، به وقوع پیوندد. طی این فرآیند، لازم است که اطلاعات ورودی به پردازنده اعمال شود؛ سپس پردازش صورت گیرد تا خروجی مناسب تولید شده و در پایان، از اطلاعت خروجی بهره‌برداری شود.

هر پردازنده، بسته به نوع، می‌تواند یک یا چند نوع پردازش بر داده‌های ورودی اعمال کند. همچنین با توجه به کاربرد و طراحی قطعه پردازنده، هر پردازش به طور خودکار، یا با دستور کاربر انجام می‌شود. آن چه که حائز اهمیت می‌باشد، بدون شک عملکرد صحیح پردازنده، صرف نظر از نوع و کاربرد آن است.

در این گزارش، به تشریح نوعی از پردازنده با عنوان «واحد محاسبه‌گر منطقی» یا "Arithmetic Logic Unit" می‌پردازیم. در این قطعه، با توجه به «حالت عملگر» یا "Operation Mode" که توسط کاربر تعیین می‌شود؛ پردازش خاصی بر داده‌ها صورت می‌پذیرد که نحوه انجام این عملیات را به تفصیل توضیح خواهیم داد.

در پایان، لازم به ذکر است که گزارش شامل سه بخش کلی است؛ در هر بخش، به تشریح نحوه ارضای خواسته‌های مسئله می‌پردازد. در هر بخش، سه قسمت اصلی وجود دارد؛ که شامل «معرفی»، «اعتبارسنجی» و «نکات جانبی» می‌باشد.

بخش اول (Task 1)

۱ – ۱ – معرفی کد vhdl و تشریح سیر تکمیلی مدار

با توجه به آن که مدار، به ازای operation code های مختلف، عملیات متفاوتی را بر داده‌های ورودی اعمال می‌کند، تصمیم بر آن شد که برای انجام هر عملیات یا یک دسته عملیات، یک Component طراحی شود. سپس در کد اصلی، هر Component را فراخوانی کرده‌ایم؛ و بعد از انتخاب نام دلخواه برای هر کدام، Port Map را مشخص کرده‌ایم. نمونه‌ای از Component های فراخوانی شده و نقشه Port Map مربوط به آن را در زیر می‌بینیم:

```
COMPONENT Unsigned_Addition IS
        PORT (
            Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            Cin : IN STD_LOGIC;
            Op : IN STD_LOGIC;
            Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
            Cout : OUT STD_LOGIC);
    END COMPONENT;
```

کد شماره ۱- فراخوانی Component مربوط به جمع‌کننده اعداد بی‌علامت "Unsigned_Addition"

```
U2 : Unsigned_Addition PORT MAP(
        Input1 => A, Input2 => B, Op => OPCODE(1), Cin => Cin,
        Output1 => X_UADD, Cout => Cout_UADD);
```

کد شماره ۲- Port Map

در پایان، تنها کاری که باید انجام می‌شد، فراخوانی هر بخش از مدار توسط Operation Code بود. برای راحتی کار، یعنی بهره‌گیری از منطق else و if ، از دستور Process استفاده کرده‌ایم؛ تا به ازای هر Operation Code مشخص، Component مناسب فعال شود. بخشی از دستور Process مربوط به جمع‌کننده اعداد بی‌علامت "Unsigned_Addition" را در زیر آورده‌ایم:

```
        -------Unsigned_Addition--------
    ELSIF OPCODE = x"4" OR OPCODE = x"6" THEN
        X <= X_UADD;
        Y <= x"00";
        Cout <= Cout_UADD;
        V <= '0';
        N <= '0';
```

کد شماره ۳- نحوه فراخوانی تابع Unsigned Addition با استفاده از Operation Code

توجه داریم که سیگنال‌های میانی تعریف شده در تکه کدهای بالا به چشم می‌خورد که در گزارش معرفی نشده است؛ گر چه که نام این سیگنال‌ها با وظیفه آن همخوانی مناسب دارد، پیشنهاد می‌شود که نحوه تعریف و کاربرد این سیگنال‌ها را در کد اصلی مشاهده فرمایید.

برای بررسی وضعیت هر Flag ، تکه کدهایی را در کد اصلی آورده‌ایم که با کامنت مشخص شده است و بعد از تشریح نحوه عملکرد هر Component ، به توضیح آن‌ها می‌پردازیم.

با رعایت نسبی ترتیب Operation Code داریم:

- **عملگر Logicals**

در این کد، با توجه به مقدار Operation Code ، یکی از اَعمال منطقی به ورودی ها اِعمال می‌شود. برای سادگی، از دستور With Select بهره برده‌ایم. توجه داریم که فقط دو بیت سمت راست Operation Code نوع عملگر را تعیین می‌کند؛ پس فقط پایه‌های ۰ و ۱ ورودی OPCODE را به عنوان ورودی کنترل این Component در نظر گرفته‌ایم.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Logicals IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Op : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END Logicals;

ARCHITECTURE Behavioral OF Logicals IS

BEGIN

    -- Calculating the result of logical operations
    -- 00 -> AND
    -- 01 -> OR
    -- 02 -> XOR
    -- 03 -> XNOR

    -- Determining Output1 for Each Operation
    WITH Op SELECT
        Output1 <= (Input1 AND Input2) WHEN "00",
        (Input1 OR Input2) WHEN "01",
        (Input1 XOR Input2) WHEN "10",
        (Input1 XNOR Input2) WHEN "11";

END Behavioral;
```

کد شماره ۴- انجام عملیات‌های And, Or, Xor, Xnor با استفاده از اپراتور و تعیین خروجی‌ها

- عملگر Unsigned Addition

واضح است که در این بخش، سیگنال ۹ بیتی Sum را تعریف کرده‌ایم تا حاصل جمع دو عدد ۸ بیتی باشد. سپس ۸ بیت اول آن را به پایه خروجی ۸ بیتی داده‌ایم و بیت آخر، در خروجی Cout ظاهر می‌شود. توجه داریم که ورودی ها نیز، با اضافه کردن یک بیت ۰ به ابتدای رشته، با سیگنال Sum هم‌طول شده است.

لازم به ذکر است که با توجه به مقدار Op ، که در واقع همان OPCODE(1) می‌باشد، مقدار Cin در حاصل جمع شرکت داده می‌شود. می‌دانیم که اگر OPCODE(1) = 0 باشد، عمل جمع بدون در نظر گرفتن Cin انجام می‌شود؛ در غیر این صورت، باید عمل جمع با لحاظ شدن Cin انجام شود. این موضوع با چک کردن مقدار OPCODE در صورت پروژه به سادگی قابل درک است.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Unsigned_Addition IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cin : IN STD_LOGIC;
        Op : IN STD_LOGIC;
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC);
END Unsigned_Addition;

ARCHITECTURE Behavioral OF Unsigned_Addition IS

    -- Defining Signals
    SIGNAL Sum : STD_LOGIC_VECTOR (8 DOWNTO 0);
    SIGNAL C : STD_LOGIC;

BEGIN

    -- Unsigned Addition: Op = '0'
    -- Unsigned Addition with Carry: Op = '1'

    -- Determining C for Each Operation
    C <= Cin WHEN Op = '1' ELSE
        '0';
```

```
    -- Calculating the Summation
    Sum <= ('0' & Input1) + ('0' & Input2) + C;

    Output1 <= Sum(7 DOWNTO 0);

    -- Determining Cout
    Cout <= Sum(8);

END Behavioral;
```

کد شماره ۵- انجام عملیات Unsigned Addition with Carry & without Carry و تعیین خروجی‌ها

- **عملگر Signed Addition**

در این بخش، با استفاده از پکیج‌های موجود در کتابخانه‌های اضافه شده، اعداد ورودی را با فرض اعداد علامت‌دار جمع کرده‌ایم. همچنین با توجه به نکته‌ای که در بخش قبل اشاره شد، سیگنال SignedAdd را ۹ بیتی تعریف کرده‌ایم. نکته‌ای که هنگام هم‌طول کردن ورودی ها و سیگنال SignedAdd حائز اهمیت است، علامت‌دار بودن اعداد ورودی می‌باشد؛ به همین علت است که بیت اضافه شده به ابتدای رشته‌های ورودی، برابر بیت شماره ۷ ورودی است. توجه داریم که نهایتا سیگنال SignedAdd را که از نوع Signed است، در سیگنال STDAns که از نوع STD_LOGIC_VECTOR می‌باشد می‌ریزیم تا به پایه‌های خروجی اعمال شود.

برای تشخیص وضعیت OverFlow ، از دستورالعملی که در جلسات ابتدایی کلاس درس برای جمع اعداد علامت‌دار ارائه شد، بهره برده‌ایم. این دستورالعمل به روشنی در کامنت‌های لحاظ شده در متن کد آمده است. در این کد، مطابق خواسته مسئله، N <= S(8) لحاظ شده است.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Signed_Addition IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        V : OUT STD_LOGIC;
        N : OUT STD_LOGIC);
END Signed_Addition;

ARCHITECTURE Behavioral OF Signed_Addition IS

    -- Defining Signals
    SIGNAL SignedAdd : SIGNED (8 DOWNTO 0);
    SIGNAL STDAns : STD_LOGIC_VECTOR (8 DOWNTO 0);

BEGIN

    -- Copying the Sign Bit to the left side of Input1 and Input2
    -- Converting datatype of Input1 and Input2 to SIGNED
    -- Calculating Signed Addition
    SignedAdd <= SIGNED(Input1(7) & Input1) + SIGNED(Input2(7) & Input2);

    -- Converting datatype of SignedAdd to Standard logic vector
    STDAns <= STD_LOGIC_VECTOR(SignedAdd);

    Output1 <= STDAns(7 DOWNTO 0);

    -- OverFlow Detection
    -- Cn = Cn-1 => Overflow = '0' => V = '0'
    -- Cn != Cn-1 => Overflow = '1' => V = '1'
    V <= '1' WHEN (SignedAdd(8) /= SignedAdd(7)) ELSE
        '0';

    -- Determining N
    N <= STDAns(8);

END Behavioral;
```

کد شماره ۶- انجام عملیات Signed Addition و تعیین خروجی‌ها

- عملگر Signed & Unsigned Multiplication

در این قسمت همانند قسمت logical ، با استفاده از سیگنال میانی Op یک اپراتور تعریف کرده‌ایم تا میان دو حالت ورودی signed و unsigned یکی را انتخاب کند. برای انجام عملیات ضرب با علامت، با فراخوانی پکیج IEEE.NUMERIC_STD.ALL، ورودی‌ها را به صورت signed تعریف می‌کنیم و سپس عملیات ضرب را انجام می‌دهیم. در پایان، پرچم N، مطابق خواسته مسئله طراحی شده است.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Multiplication IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Op : IN STD_LOGIC;
        Output1, Output2 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        N : OUT STD_LOGIC);
END Multiplication;

ARCHITECTURE Behavioral OF Multiplication IS

    -- Defining Signals
    SIGNAL SignedMult : SIGNED (15 DOWNTO 0);
    SIGNAL MultAns : STD_LOGIC_VECTOR (15 DOWNTO 0);

BEGIN
    SignedMult <= SIGNED(Input1) * SIGNED(Input2);

    -- Unsigned Multiplication: Op --> '1'
    -- Signed Multiplication: Op --> '0'

    -- Determining MultAns for Each Operation
    MultAns <= Input1 * Input2 WHEN Op = '1' ELSE
        STD_LOGIC_VECTOR(SignedMult);

    -- Slicing MultAns to Output1 And Output2
    Output1 <= MultAns(7 DOWNTO 0);
    Output2 <= MultAns(15 DOWNTO 8);
```

```
    -- N = '0' for Unsigned Multiplication
    -- N = MultAns(15) for Signed Multiplication

    -- Determining N for Each Operation
    N <= MultAns(15) WHEN Op = '0' ELSE
        '0';

END Behavioral;
```

کد شماره ۷- تعیین خروجی‌های ماژول Signed & Unsigned Multiplication

- **عملگر Unsigned Subtraction**

این کد، مشابه کد جمع بدون علامت است. از اسلایدهای ابتدایی کلاس، نحوه تفریق دو عدد باینری بدون علامت را به خاطر داریم؛ فرآیندی که به وضوح در کامنت‌های ارائه شده در کد شماره ۸ تشریح شده است. لازم به ذکر است که برای استفاده از منطق if و else، از دستور Process بهره برده‌ایم تا با توجه به مقدار کری خروجی یا همان Sum(8)، حاصل تفریق را به پایه‌های خروجی و Cout اعمال کنیم.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Unsigned_Subtraction IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC);
END Unsigned_Subtraction;

ARCHITECTURE Behavioral OF Unsigned_Subtraction IS
    SIGNAL Sum : STD_LOGIC_VECTOR (8 DOWNTO 0);

BEGIN
    Sum <= ('0' & Input1) + ('0' & NOT(Input2));
   -- If Input1 >= Input2  => Sum(8) = '1' => Answer is positive.(Cout = '0')
    --           => End Carry => This end carry should be ignored.
```

```
    -- If Input1 < Input2   => Sum(8) = '0' => Answer is negative.(Cout = '1')
    --           => Ans = 1's Complement of Sum with a minus sign
    PROCESS (Sum(8), Sum)
    BEGIN
        IF (Sum(8) = '1') THEN
            Output1 <= Sum(7 DOWNTO 0) + x"01";
            Cout <= '0';
        ELSE
            Output1 <= NOT (Sum(7 DOWNTO 0));
            Cout <= '1';
        END IF;
    END PROCESS;
END Behavioral;
```

کد شماره ۸- انجام عملیات Unsigned Subtraction و تعیین خروجی‌ها

- عملگر **Rotation Left with & without Carry**

در این Component با توجه به مقدار Op که سیگنالی میانی است، میان دو عملگر اشاره شده در تیتر این بخش، انتخاب می‌شود. توجه داریم که اگر عملگر **Rotation left without carry** با Op = 0 فعال شود، Cout = 0 خواهد بود؛ در غیر این صورت و به ازای Op = 1 ، بیت شماره هفت ورودی به Cout اعمال می‌شود.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Rotation IS
    PORT (
        Input1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Op : IN STD_LOGIC;
        Cin : IN STD_LOGIC;
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC);
END Rotation;
```

```
ARCHITECTURE Behavioral OF Rotation IS

    -- Defining Signals
    SIGNAL LSB : STD_LOGIC;

BEGIN

    -- Rotation Left: Op = '0'
    -- Rotation Left with Carry: Op = '1'

    -- Determining LSB for Each Operation
    LSB <= Input1(7) WHEN Op = '0' ELSE
        Cin;

    Output1 <= Input1(6 DOWNTO 0) & LSB;

    -- Determining Cout for Each Operation
    Cout <= '0' WHEN Op = '0' ELSE
        Input1(7);

END Behavioral;
```

کد شماره ۹- انجام عملیات Rotation left با Carry و بدون Carry

- **عملگر Logic & Arithmetic Shift Right**

مطابق توضیحات ارائه شده در بخش قبل، با توجه به مقدار سیگنال میانی Op ،یکی از دو عملگر این Component فعال می‌شود. همان‌طور که می‌دانیم، با فعال شدن Logic Shift Right، مقدار MSB برابر ۰ می‌شود؛ اما در Arithmetic Shift Right، این مقدار برابر با بیت سمت چپ ورودی می‌باشد. در هر دو عملگر، تمام بیت‌ها، قبل از تغییر مقدار MSB، یک واحد به سمت راست حرکت می‌کند. خروجی Cout نیز برابر با بیت سمت راست ورودی است.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Shift_Right IS
    PORT (
        Input1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Op : IN STD_LOGIC;
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC;
        N : OUT STD_LOGIC);
END Shift_Right;

ARCHITECTURE Behavioral OF Shift_Right IS
    -- Defining Signals
    SIGNAL MSB : STD_LOGIC;

BEGIN

    -- Logic Shift Right: Op = '0'
    -- Arithmetic Shift Right: Op = '1'

    -- Determining MSB for Each Operation
    MSB <= '0' WHEN Op = '0' ELSE
        Input1(7);

    Output1 <= MSB & Input1(7 DOWNTO 1);
    N <= MSB;

    -- Determining Cout
    Cout <= Input1(0);

END Behavioral;
```

کد شماره ۱۰- تعیین خروجی *Logical & Arithmetic Shift Right*

- عملگر **Shift Left**

این عملگر همانند Logical Shift Right بر ورودی اعمال می‌شود؛ با این تفاوت که به‌جای MSB ، مقدار LSB برابر ۰ می‌شود. توجه داریم که قبل از صفر شدن LSB، تمام بیت‌های بعدی یک واحد به سمت چپ حرکت می‌کنند. خروجی Cout نیز همان بیت سمت چپ یا MSB ورودی خواهد بود.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Shift_Left IS
    PORT (
        Input1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC;
        N : OUT STD_LOGIC);
END Shift_Left;

ARCHITECTURE Behavioral OF Shift_Left IS

BEGIN

    Output1 <= Input1(6 DOWNTO 0) & '0';

    -- Determining N
    N <= Input1(6);

    -- Determining Cout
    Cout <= Input1(7);

END Behavioral;
```

کد شماره ۱۱- تعیین خروجی *Shift Left*

- عملگر BCD to Binary

در این کد، ابتدا عدد BCD را در مبنای ۱۰ می‌نویسیم. برای مثال عدد BCD به صورت "2453" برابر «دو چهار پنج سه» خوانده می‌شود و در مبنای ۱۰ به صورت «دوهزار و چهارصد و پنجاه و سه» است. برای انجام این کار، هر عدد BCD را که به فرم باینری دریافت می‌شود، به اعدادی در مبنای ۱۰ تبدیل می‌کنیم. حال هر عدد در ارزش مکانی خود ضرب می‌شود و مجموع این حاصل‌ضرب‌ها، عدد موردنظر را می‌سازد. حال کافی است که عدد نهایی را در مبنای ۲ بنویسیم. این امر با کمک دستور to_unsigned انجام می‌شود.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY BCD2BIN IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Output1, Output2 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC);
END BCD2BIN;

ARCHITECTURE Behavioral OF BCD2BIN IS
    -- Defining Signals
    SIGNAL Digit1Dec : INTEGER := 0;
    SIGNAL Digit2Dec : INTEGER := 0;
    SIGNAL Digit3Dec : INTEGER := 0;
    SIGNAL Digit4Dec : INTEGER := 0;
    SIGNAL DecNum : INTEGER := 0;
    SIGNAL BinNum : STD_LOGIC_VECTOR (15 DOWNTO 0);

BEGIN
    -- Extracting Each Digit
    Digit1Dec <= conv_integer(Input1(3 DOWNTO 0));
    Digit2Dec <= conv_integer(Input1(7 DOWNTO 4));
    Digit3Dec <= conv_integer(Input2(3 DOWNTO 0));
    Digit4Dec <= conv_integer(Input2(7 DOWNTO 4));
    -- Calculating Decimal Number
    DecNum <= Digit1Dec + (Digit2Dec * 10) + (Digit3Dec * 100)
              + (Digit4Dec * 1000);
```

```
    -- Converting Decimal Number to Binary Number
    BinNum <= STD_LOGIC_VECTOR(to_unsigned(DecNum, 16));
    -- Slicing the Binary Number to Output1 and Output 2
    Output1 <= BinNum(7 DOWNTO 0);
    Output2 <= BinNum(15 DOWNTO 8);
    -- Setting Cout with MSB
    Cout <= BinNum(15);
END Behavioral;
```

کد شماره ۱۲- تبدیل BCD به Binary

Z Flag -

این پرچم حاصل یک گیت And می‌باشد. در صورتی که خروجی‌های X و Y به فرم X = Y = "00000000" یا به عبارتی برابر ۰ باشد، همچنین Cout = 0 شود، بالا می‌رود؛ یعنی Z = 1 می‌شود. در غیر این‌صورت، Z = 0 است.

```
    --------Z FLAG-------------
    Z <= '1' WHEN (X = x"00" AND Y = x"00" AND Cout = '0') ELSE
        '0';
```

flag 1- Z flag

F_active Flag -

این پرچم حاصل Or سه خروجی است و به فرم زیر تعریف می‌شود:

```
    ---------F_active------------
    F_active <= Z OR V OR Cout;
```

flag 2- F_active flag

- X_bin_pal Flag

این پرچم حاصل تقارن افقی بیت‌های خروجی X است. به این ترتیب که اگر مقدار بیت شماره n و 7-n به طوری که $0 \leq n \leq 7$ باشد، برابر شود؛ آن گاه X_bin_pal = 1 می‌شود. با توجه به این که می‌دانیم برابری دو بیت با گیت منطقی Xnor مشخص می‌شود، کد تشخیص این پرچم را به فرم زیر نوشته‌ایم:

```
---------X_bin_pal----------
PAL(0) <= (X(0) XNOR X(7));
PAL(1) <= (X(1) XNOR X(6));
PAL(2) <= (X(2) XNOR X(5));
PAL(3) <= (X(3) XNOR X(4));
X_bin_pal <= '1' WHEN PAL = x"f" ELSE
    '0';
```

flag 3- X_bin_pal flag

- X_prime Flag

این پرچم زمانی مقدار می‌گیرد که خروجی X عدد اول باشد برای این‌کار ابتدا X را به یک ورودی Integer تبدیل می‌کنیم سپس شرط اول بودن آن را بررسی می‌کنیم. برای این کار، لازم است که مقدار باقی مانده تقسیم X بر اعداد اول کوچکتر از $\sqrt{X}$ را بررسی کنیم. بیشترین مقدار "11111111" = X برابر 255 = X است. پس کافی است باقی‌مانده X را بر اعداد اول کوچکتر از ... ،15 بررسی کنیم. همچنین به عنوان حالت خاص، باید بررسی کنیم که آیا X برابر اعداد اول کوچک‌تر از $\sqrt{X}$ می‌شود؟ فرآیندی که تشریح شد، در کد زیر مشهود است:

```
--------X_prime-----------
PROCESS (X)

    VARIABLE Number : INTEGER;

BEGIN

    Number := conv_integer(X);

    IF (Number = 0 OR Number = 1) THEN
        X_prime <= '0';
```

```
    ELSIF (Number = 2 OR Number = 3 OR Number = 5 OR Number = 7
           OR Number = 11 OR Number = 13) THEN
        X_prime <= '1';

    ELSIF (Number REM 2 = 0 OR
        Number REM 3 = 0 OR
        Number REM 5 = 0 OR
        Number REM 7 = 0 OR
        Number REM 11 = 0 OR
        Number REM 13 = 0) THEN
        X_prime <= '0';

    ELSE
        X_prime <= '1';

    END IF;
END PROCESS;
```

flag 4- X_prime flag

در پایان این بخش، یادآور می‌شویم که پرچم‌های V ، N و Cout به طور جداگانه در هر Component و با توجه به خواسته مسئله طراحی شده است. در عین حال، پرچم‌های دیگر را به طور جداگانه و در کد اصلی طراحی کرده‌ایم تا مدار بهینه تر و با تعداد گیت منطقی کمتر ساخته شود.

۱ – ۲ – بررسی آزمایش test bench و اعتبارسنجی کد vhdl

در این بخش، با ارائه نتایج تست بنچ از صحت کد نوشته شده اطمینان حاصل می‌کنیم. قبل از بررسی نتایج، فرم کلی کد تست بنچ را مورد بررسی قرار می‌دهیم.

برای تست بنچ، لازم است که یک Entity بدون ورودی و خروجی تعریف کنیم؛ سپس ماژول اصلی را به عنوان یک Component فراخوانی کرده و نقشه Port Map برای آن تعیین می‌کنیم. در پایان با استفاده از دستور Process، مقادیر دلخواهی برای ورودی در نظر می‌گیریم و خروجی را بررسی می‌کنیم.

آن چه که در پاراگراف بالا تشریح شد، به طور اجمالی در کد زیر نمایش داده شده است: (توجه داریم که برای پرهیز از شلوغی گزارش، تمام OpCode های تست شده در تکه کد زیر نیامده است. لطفا برای مشاهده تمامی OpCode ها، به فایل tb_ALU.vhd مربوط به این بخش مراجعه فرمایید.)

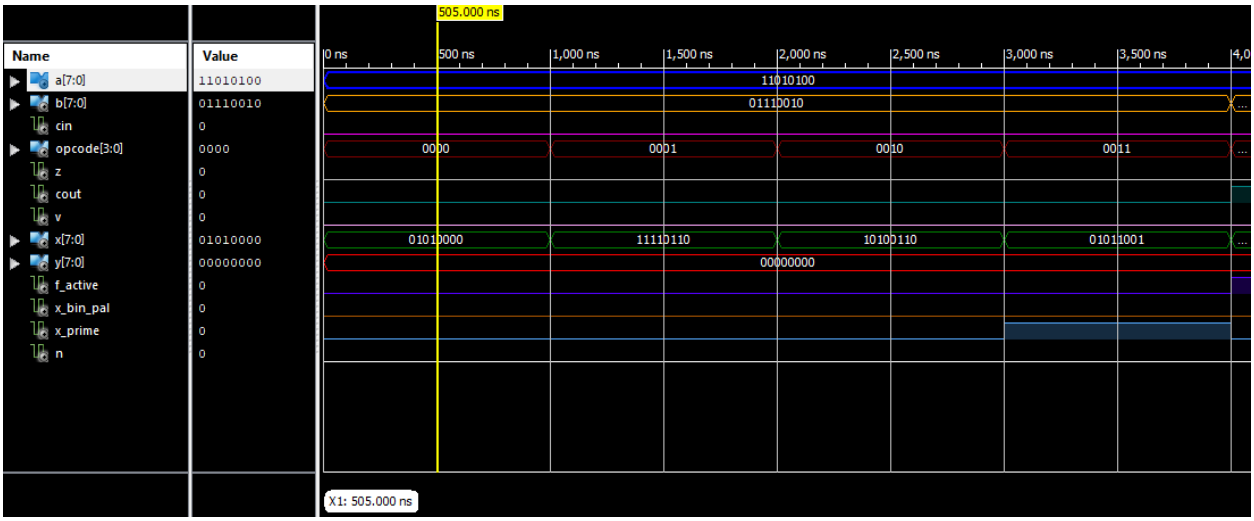
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY tb_ALU IS
END tb_ALU;

ARCHITECTURE behavior OF tb_ALU IS
    COMPONENT ALU
        PORT (
            A : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            B : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            Cin : IN STD_LOGIC;
            OPCODE : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            X : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            Y : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            Z : INOUT STD_LOGIC;
            Cout : INOUT STD_LOGIC;
            V : INOUT STD_LOGIC;
            F_active : OUT STD_LOGIC;
            X_bin_pal : OUT STD_LOGIC;
            X_prime : OUT STD_LOGIC;
            N : OUT STD_LOGIC
        );
    END COMPONENT;
```

```
    -- Defining Signals
    SIGNAL A : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
    SIGNAL B : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
    SIGNAL Cin : STD_LOGIC := '0';
    SIGNAL OPCODE : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
    SIGNAL Z : STD_LOGIC;
    SIGNAL Cout : STD_LOGIC;
    SIGNAL V : STD_LOGIC;
    SIGNAL X : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL Y : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL F_active : STD_LOGIC;
    SIGNAL X_bin_pal : STD_LOGIC;
    SIGNAL X_prime : STD_LOGIC;
    SIGNAL N : STD_LOGIC;

BEGIN

    -- Assigning ALU PORTs to the Related Signals
    uut : ALU PORT MAP(
        A => A,
        B => B,
        Cin => Cin,
        OPCODE => OPCODE,
        X => X,
        Y => Y,
        Z => Z,
        Cout => Cout,
        V => V,
        F_active => F_active,
        X_bin_pal => X_bin_pal,
        X_prime => X_prime,
        N => N
    );

    -- Activating Stimulation Process
    stim_proc : PROCESS
    BEGIN
        --OPCODE1 : AND
        opcode <= x"0";
        A <= x"d4";
        B <= x"72";
        Cin <= '0';
        WAIT FOR 1 us;
```

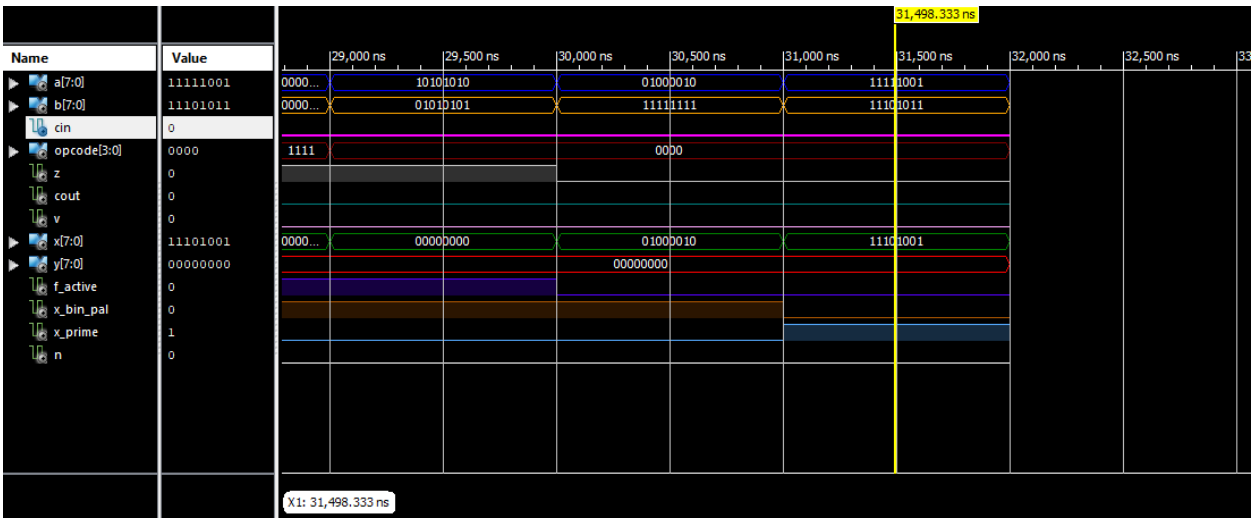
```
        -- OPCODE2:  OR
        opcode <= x"1";
        A <= x"d4";
        B <= x"72";
        Cin <= '0';
        WAIT FOR 1 us;

        .
        .
        .

        -- Testing X_prime
        opcode <= x"0";
        A <= x"f9";
        B <= x"eb";
        Cin <= '0';
        WAIT FOR 1 us;

        WAIT;
    END PROCESS;

END;
```

کد شماره ۱۳- Test Bench

اکنون دو مورد از نتایج آزمایش تست‌بنچ را ارائه می‌کنیم. توجه داریم که ارائه تمامی نتایج تست‌بنچ در گزارش، موجب شلوغی و ایجاد فرم نامطلوب می‌شود. خواهشمند است که نتایج آزمایش باقی Component ها را که به صورت فایل تصویری در پوشه Task1TestBenchSCs قرار داده‌ایم، مشاهده فرمایید. ترتیب نام‌گذاری نتایج آزمایش تست‌بنچ برای هر Component در این پوشه، مطابق مقدار operation code برای هر عملیات است.



شکل ۱- نتیجه تست عملگر Logicals



شکل ۲- نتیجه تست پرچم X_prime (به ازای عملگر And)

$$(11101001)_2 = 2^7 + 2^6 + 2^5 + 2^3 + 2^0 = 233\ is\ a\ prime\ number$$

۱ – ۳ – نکات جانبی پیرامون نتایج آزمایش

در این بخش تلاش می‌کنیم تا به نکاتی حاشیه‌ای بپردازیم که می‌تواند جالب توجه باشد.

اول: جالب است بدانید که اگرچه پرچم Cout نسبت به خروجی عملگر BCD to Binary حساس است، اما به ازای هر ورودی، Cout = 0 باقی خواهد ماند.

می‌دانیم که بزرگترین ورودی این عملگر به صورت زیر است:

$$(10011001 : 10011001)_{BCD} = (9999)_{10} = (00100111 : 00001111)_{Binary}$$

به طوری که ۸ بیت اول، در خروجی Y ظاهر می‌شود. از طرفی Cout = Y(7) می‌شود. وقتی به ازای بیشترین مقدار ورودی ممکن Y(7) = 0 شود، پس همواره Cout = 0 خواهد شد.

دوم: با توجه به جدول ارائه شده در پروژه، پرچم V به عملگر Logic Shift Left حساسیت دارد؛ در صورتی که این عملیات ارتباطی با بحث OverFlow ندارد. به همین خاطر در پایان مباحثه پیرامون این قضیه، به این نتیجه رسیدیم که حساسیت این پرچم هنگام انجام عملیات Logic Shift Left ، سهوا در صورت پروژه ثبت شده است؛ پس از آن صرف نظر کرده‌ایم. این تصمیم، به وضوح در کد vhdl ارائه شده مشهود است.

سوم: شمای کلی مدار ساخته شده، تحت عنوان فایل pdf به نام Schematic Task1 به فایل‌های پروژه ضمیمه شده است.

بخش دوم (Task 2)

۱ – ۲ – معرفی کد vhdl و تشریح سیر تکمیلی مدار

در این بخش، تمام Component های معرفی شده در Task 1، دقیقا همان عملکرد را دارند؛ پس با تعریف Component ALU مدار طراحی شده را در قالب یک Component فراخوانی کرده و فقط نحوه تأثیر ورودی‌های Clock و Reset را در خروجی مدار بررسی می‌کنیم. به عبارتی تلاش می‌کنیم تا ALU را که در قسمت قبل طراحی شد، به سیستم Controller Logic مجهز کنیم. در ادامه به طراحی Port Map برای مدار جدید می‌پردازیم.

```
COMPONENT ALU IS
     PORT (
         A : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
         B : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
         Cin : IN STD_LOGIC;
         OPCODE : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
         X : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
         Y : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
         Z : INOUT STD_LOGIC;
         Cout : INOUT STD_LOGIC;
         V : INOUT STD_LOGIC;
         X_bin_pal : OUT STD_LOGIC;
         X_prime : OUT STD_LOGIC;
         N : OUT STD_LOGIC);
 END COMPONENT;
```

کد شماره ۱۴- نحوه تعریف تابع ALU

```
    Unit : ALU PORT MAP(
        A => A_ALU, B => B_ALU, Cin => Cout, OPCODE => OPCODE, X => X_ALU, Y =>
        Y_ALU, Z => Z_ALU,
        Cout => Cout_ALU, V => V_ALU, X_bin_pal => X_bin_pal_ALU, X_prime => X_
prime_ALU, N => N_ALU);
```

کد شماره ۱۵- ALU Port Map

در ابتدا، باید ورودی‌های مدار ترتیبی ALU را مشخص کنیم. برای این کار از یک DeMultiplexer 1 to 2 استفاده می‌کنیم و با توجه به مقدار SEL_IN ، ورودی I را به شیفت رجیسترها وارد می‌کنیم. توجه داریم که شیفت رجیسترها به Rising_Edge(CLK) و مقدار Load حساس است؛ به طوری که به ازای Load=1، فعال می‌شود و ورودی جدید را دریافت می‌کند. اما قبل از هر چیز، مقدار ورودی Reset بررسی می‌شود؛ چرا که فرآیند بازنشانی مدار از نوع Asynchrone است. لازم به ذکر است که به ازای Reset = 1 ، تمام بیت های شیفت رجیسترهای A و B برابر صفر می‌شوند.

```
-- Input Registers Process
PROCESS (SEL_IN, I, CLK, LOAD, RESET)
  BEGIN
      IF (RESET = '1') THEN
          A_ALU <= x"00";
          B_ALU <= x"00";

      ELSIF rising_edge(CLK) AND LOAD = '1' THEN
          IF (SEL_IN = '0') THEN
              A_ALU <= I;
          ELSIF (SEL_IN = '1') THEN
              B_ALU <= I;
          END IF;
      END IF;
  END PROCESS;
```

کد شماره ۱۶- نحوه عملکرد بخش ابتدایی مدار برای تعیین ورودی‌های ALU

بعد از تبیین ورودی‌ها، به معرفی خروجی‌های مدار می‌پردازیم. مدار را به نحوی طراحی کرده‌ایم تا به ازای Reset = 1 ، تمام خروجی‌ها برابر صفر باشد. در غیر این‌صورت با دستور Rising_Edge(clk) وارد بدنه اصلی کد می‌شویم.

```
PROCESS (RESET, CLK, OPCODE, RUN, SEL_OUT, Cout, V, Z, en_C, en_V, en_N, X_ALU,
 Y_ALU, Z_ALU,
       Cout_ALU, N_ALU, V_ALU, X_bin_pal_ALU, X_prime_ALU)

    BEGIN
        -- Output Registers Dependent on OPCODE and Reset
        IF (RESET = '1') THEN
            R <= (OTHERS => '0');
            Cout <= '0';
            Z <= '0';
            V <= '0';
            N <= '0';
            X_prime <= '0';
            X_bin_pal <= '0';

        ELSIF rising_edge(CLK) THEN
        .
        .
        .
```

کد شماره ۱۷- اجازه دسترسی ALU برای تولید خروجی و حساسیت آن به Rising_Edge

حال نحوه تأثیر هر عملیات یا "operation" را بر خروجی‌های مدار بررسی می‌کنیم. برای جلوگیری از شلوغ شدن گزارش، در این بخش فقط دو مورد از عملیات اشاره شده را بررسی کرده‌ایم، خواهشمند است که نحوه فعال‌سازی دیگر عملگرها را در فایل ALU_SEQ.vhd مشاهده کنید.

```
    -- Logicals
     IF OPCODE = x"0" OR OPCODE = x"1" OR OPCODE = x"2" OR OPCODE = x"3" THEN
                en_N <= '1';
                en_V <= '0';
                en_C <= '0';
```

کد شماره ۱۸- تغییر Flag ها بر اثر تابع Logicals از ALU

```
        -- Rotation Left with Carry
        -- Logic Shift Right
        -- Arithmetic Shift Right
        -- Logic Shift Left
ELSIF OPCODE = x"b" OR OPCODE = x"c" OR OPCODE = x"d" OR OPCODE = x"e" THEN
            en_N <= '1';
            en_V <= '0';
            en_C <= '1';
```

کد شماره ۱۹- تغییر Flag ها بر اثر توابع ذکر شده به صورت comment از ALU

همان‌طور که می‌بینیم در هر تکه کد، تأثیر هر عملگر فقط بر پرچم‌های en_N و en_C و en_V ذکر شده‌است؛ زیرا این مقدار این سیگنال‌ها، به ورودی OpCode بستگی دارد. با توجه به Table-2 که در صورت پروژه آمده است، اگر عملیات بر روی خروجی تاثیر گذار باشد، مقدار enable خروجی برابر ۱ می‌شود تا مقدار محاسبه شدهٔ جدید وارد خروجی شود؛ در غیر این‌صورت، enable مقدار ۰ می‌گیرد و مقدار خروجی بدون تغییر باقی می‌ماند.

```
        -- FlipFlop for Activating Cout
        IF en_C = '1' THEN
            Cout <= Cout_ALU;
        END IF;
        -- FlipFlop for Activating V
        IF en_V = '1' THEN
            V <= V_ALU;
        END IF;
        -- FlipFlop for Activating N
        IF en_N = '1' THEN
            N <= N_ALU;
        END IF;
```

کد شماره ۲۰- نحوه ایجاد Flagهای ذکر شده در خروجی مدار

در ادامه، باقی خروجی‌ها و Flag های مدار را با کمک خروجی‌های ALU و دستور Run تعیین می‌کنیم. با فعال شدن Run ، تمام خروجی‌های ALU وارد خروجی‌های مدار می‌شود. برای تعیین خروجی R از یک Multiplexer 2 to 1 استفاده می‌کنیم که SEL_OUT بین X_ALU و Y_ALU یکی را انتخاب کرده و وارد خروجی R می‌کند.

```
-- FlipFlop for Activating Outputs via RUN
IF RUN = '1' THEN
    Z <= Z_ALU;
    X_bin_pal <= X_bin_pal_ALU;
    X_prime <= X_prime_ALU;
    IF (SEL_OUT = '0') THEN
        R <= X_ALU;
    ELSIF (SEL_OUT = '1') THEN
        R <= Y_ALU;
    END IF;
END IF;
```

کد شماره ۲۱- روشن شدن Run و تعیین خروجی‌ها و Flagهای مدار

تنها Flag که باقی مانده F_active است که آن را به فرم زیر تعریف کرده‌ایم:

```
F_Active <= Cout OR V OR Z;
```

کد شماره ۲۲- F_active

۲ – ۲ – بررسی آزمایش test bench و اعتبارسنجی کد vhdl

فرم کلی نحوه تعریف تست بنچ و بهره‌برداری از آن، مفصلا در بخش ۱-۲ تشریح شد. اکنون توجه شما را به بخشی از کد تست‌بنچ در نظر گرفته شده برای بخش ترتیبی مدار یا "Task 2" جلب می‌کنیم: (همانطور که در بخش ۱-۲ نیز اشاره کردیم، برای پرهیز از شلوغی، از ارائه کامل کد در گزارش خودداری کرده‌ایم. خواهشمند است که برای مشاهده کامل کد، به فایل tb_ALU_SEQ.vhd مراجعه فرمایید.)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY tb_ALU_SEQ IS
END tb_ALU_SEQ;

ARCHITECTURE Structral OF tb_ALU_SEQ IS

    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT ALU_SEQ
        PORT (
            I : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            LOAD : IN STD_LOGIC;
            RESET : IN STD_LOGIC;
            SEL_IN : IN STD_LOGIC;
            SEL_OUT : IN STD_LOGIC;
            RUN : IN STD_LOGIC;
            CLK : IN STD_LOGIC;
            OPCODE : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            Z : INOUT STD_LOGIC;
            Cout : INOUT STD_LOGIC;
            V : INOUT STD_LOGIC;
            N : INOUT STD_LOGIC;
            R : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            F_active : OUT STD_LOGIC;
            X_bin_pal : OUT STD_LOGIC;
            X_prime : OUT STD_LOGIC
        );
    END COMPONENT;
```

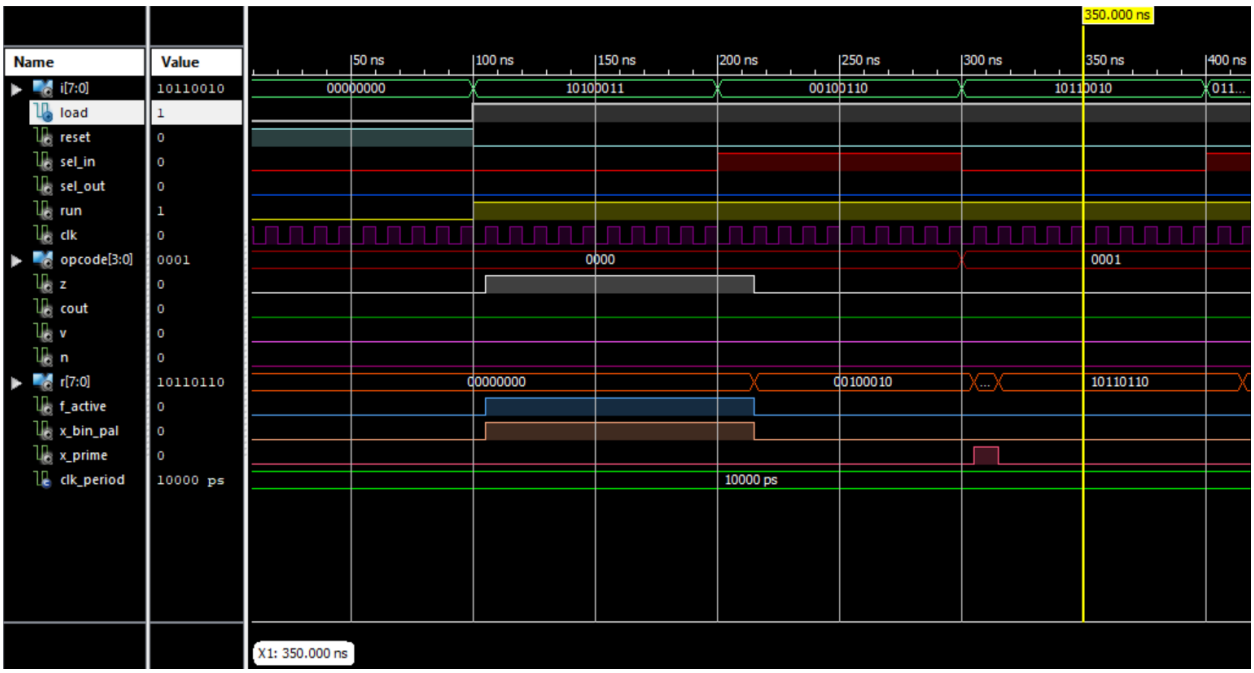
```
    -- Defining Input Signals
    SIGNAL I : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
    SIGNAL LOAD : STD_LOGIC := '0';
    SIGNAL RESET : STD_LOGIC := '0';
    SIGNAL SEL_IN : STD_LOGIC := '0';
    SIGNAL SEL_OUT : STD_LOGIC := '0';
    SIGNAL RUN : STD_LOGIC := '0';
    SIGNAL CLK : STD_LOGIC := '0';
    SIGNAL OPCODE : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
    -- Bidirectional Buses
    SIGNAL Z : STD_LOGIC;
    SIGNAL Cout : STD_LOGIC;
    SIGNAL V : STD_LOGIC;
    SIGNAL N : STD_LOGIC;
    -- Defining Output Signals
    SIGNAL R : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL F_active : STD_LOGIC;
    SIGNAL X_bin_pal : STD_LOGIC;
    SIGNAL X_prime : STD_LOGIC;
    -- Clock Period Definitions
    CONSTANT CLK_period : TIME := 10 ns;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    UUT : ALU_SEQ PORT MAP(
        I => I,
        LOAD => LOAD,
        RESET => RESET,
        SEL_IN => SEL_IN,
        SEL_OUT => SEL_OUT,
        RUN => RUN,
        Clk => Clk,
        OPCODE => OPCODE,
        R => R,
        Z => Z,
        Cout => Cout,
        V => V,
        F_active => F_active,
        X_bin_pal => X_bin_pal,
        X_prime => X_prime,
        N => N
    );
```

```
    -- Clock process definitions
    CLK_process : PROCESS
    BEGIN
        CLK <= '0';
        WAIT FOR CLK_period/2;
        CLK <= '1';
        WAIT FOR CLK_period/2;
    END PROCESS;

    -- Stimulation Process
    stim_proc : PROCESS
    BEGIN
        ------------------- Time: 0 - 100ns
        -- Initial Rest
        Reset <= '1';
        WAIT FOR 100 ns;

        -------------------- Time: 100 - 200ns
        -- Enabling
        Reset <= '0';
        Load <= '1';
        Run <= '1';

        ------------------------------------- U2 : OR
        -------------------- Time: 300 - 400ns
        OPCODE <= x"1";

        I <= x"b2";
        sel_in <= '0';
        WAIT FOR 100 ns;
.
.
        ------------------------------------- Testing Reset
        ------------------- Time: 3600 - 3700ns
        Reset <= '1';
        WAIT FOR 100 ns;

        WAIT;

    END PROCESS;
END;
```

کد شماره ۲۳- Test Bench مربوط به بخش ALU_SEQ

اکنون یک مورد از نتایج آزمایش تست‌بنچ را بررسی می‌کنیم. مطابق بخش ۱-۲ ، برای پرهیز از شلوغی، از ارائه تمام نتایج تست‌بنچ در گزارش خودداری می‌کنیم. خواهشمند است که نتایج باقی آزمایش‌ها را که به صورت فایل تصویری در پوشه Task2TestBenchSCs قرار داده‌ایم، مشاهده فرمایید. ترتیب نام‌گذاری نتایج آزمایش تست‌بنچ در این پوشه، مطابق مقدار operation code و بازه زمانی آزمایش برای هر عملیات است.

تصمیم داریم نتایج آزمایش، به ازای بازه 300ns تا 400ns را بررسی کنیم؛ دلیل این انتخاب، رخ دادن نتایجی در این بازه است که شاید در نگاه اول خاص و غیرمنتظره به نظر برسد. به هر حال تلاش می‌کنیم تا موضوع را روشن کینم. نتیجه آزمایش، با توجه به مقدار ورودی‌ها و مقدار Operation Code ، به فرم زیر است:



شکل ۳- نتایج Test Bench

برای شروع، به حساسیت پرچم‌ها به خروجی R و کلاک سیستم اشاره می‌کنیم. برای مثال در تصویر بالا، به لحظه 100ns دقت کنید؛ هر پرچم با توجه به آنکه توسط شرط لازم خود ارضا شده باشد، با رسیدن اولین لبه بالا رونده کلاک، مقداری را که باید اتخاذ می‌کند. این موضوع، با حساسیت فلیپ فلاپ‌های طراحی شده برای هر پرچم به لبه بالارونده کلاک سیستم، به سادگی قابل توجیه است.

نکته دوم پیرامون پرچم‌ها، حساسیت آن به لحظه اتمام مقداری مشخص برای خروجی R است. اگر به محور زمان دقت کنیم، به محض تغییر خروجی R ، پرچم‌ها در صورت لزوم تغییر می‌کند. این مورد، با سیگنال Enable

وارد شده به هر فلیپ فلاپ سازنده پرچم‌ها ارتباط دارد. کافی است به یکسان بودن سیگنال Run ، به عنوان سیگنال Enable ، فلیپ فلاپ ها دقت کنیم. می‌بینیم که سیگنال Enable شیفت رجیسترهای X و Y ، که تأثیر مستقیم بر خروجی R می‌گذارد، همان سیگنال Run می‌باشد.

در ادامه به حساسیت خروجی R ، به کلاک سیستم می‌پردازیم. می‌بینیم که با تغییر ورودی، مقدار معتبر خروجی R ، پس از دو دوره کلاک حاصل می‌شود. این مسئله به سادگی و با توجه به فرم مدار قابل توجیه است. وقتی ورودی I وارد سیستم شود، به اندازه یک دوره کلاک در شیفت رجیستر های نگه‌دارنده A یا B تأخیر دارد. در مرحله بعد و پس از انجام عملیات لازم بر این سیگنال توسط ALU ، سیگنال حاصل شده باز هم به اندازه یک دوره کلاک در شیفت رجیستر های نگه‌دارنده X یا Y تأخیر می‌خورد. به این ترتیب است که خروجی معتبر R ، پس از رسیدن دومین لبه بالارونده کلاک از زمان اعمال ورودی، حاصل می‌شود.

در ادامه تشریح نتایج، توجه شما را به مقدار نامعتبر R که اندکی پس از لحظه 300ns حاصل شده است، جلب می‌کنیم. همانطور که در پراگراف بالا توضیح دادیم، برای حاصل شدن خروجی معتبر، به اندازه دو دوره کلاک زمان نیاز داریم. اما در میان این بازه، یک بار شاهد لبه بالارونده کلاک هستیم. پس به سادگی توجیه می‌کنیم که: به ازای اولین لبه بالا رونده کلاک، ورودی ALU ، یعنی A یا B (با توجه به مقدار SEL_IN) تغییر می‌کند. از طرفی اولین لبه بالا رونده کلاک، سبب ذخیره‌سازی مقادیری است که در شیفت رجیستر های نگه‌دارنده X یا Y می‌باشد. پس در عین تغییر A یا B و قبل از اعمال مقدار نهایی آن در شیفت رجیسترها ، خروجی R که متأثر از مقدار X و Y است، تغییر می‌کند؛ این مقدار نامعتبر، به اندازه یک دوره کلاک در شیفت رجیستر مربوطه ذخیره می‌شود. در پایان اشاره می‌کنیم که قبل از رسیدن لبه بالارونده دوم در کلاک، ورودی I ، در شیفت رجیستر های نگه‌دارنده A یا B تثبیت شده است. پس با رسیدن این لحظه، عملیات لازم که از قبل توسط ALU بر ورودی تثبیت شده اعمال شده است، توسط شیفت رجیستر X یا Y ، به خروجی R می‌رسد تا مقدار معتبر آن را ارائه کند.

به عنوان نکته پایانی، گر چه که بدیهی است، اشاره می‌کنیم که نتیجه حاصل شده به ازای OpCode = 1 ، برابر عمل منطقی A And B می‌باشد. اگر به تغییرات متغیر SEL_IN توجه کنیم، می‌دانیم که در بازه 200ns تا 300ns ، مقدار I وارد شیفت رجیستر B شده است؛ و بعد از آن با تغییر SEL_IN ، مقدار B دست نخورده باقی می‌ماند. در بازه 300ns تا 400ns ، مقدار I وارد شیفت رجیستر A می‌شود. پس بعد از زمان گذار (مطابق توضیحات بالا، برابر دو دوره کلاک)، حاصل R ، برابر عمل منطقی And ، با ورودی‌های "A = I ; 300ns to 400ns" و "B = I ; 200ns to 300ns" می‌باشد.

۲ – ۳ – نکات جانبی پیرامون نتایج آزمایش

به عنوان نکات حاشیه‌ای در این بخش، موارد زیر را مطرح می‌کنیم:

اول: در شمای کلی ارائه شده در صورت پروژه، ورودی en0 و en1 ارائه شده است که در جدول توضیحات مطرح نمی‌شود. این ورودی ها را برابر Load در نظر گرفته‌ایم. همچنین یک مالتی پلکسر در ورودی قرار دارد که پایه انتخاب آن، SEL_IN می‌باشد. این مالتی پلکسر، ورودی I را به یکی از ورودی‌های A یا B در ALU منتقل می‌کند و در شمای اشاره شده نیامده است. در طراحی انجام شده توسط این گروه، دو نکته بالا در نظر گرفته شده است.

دوم: شمای کلی مدار ساخته شده، تحت عنوان فایل pdf به نام Schematic Task2 به فایل‌های پروژه ضمیمه شده است.

بخش امتیازی (Bonus Task)

۳ – ۱ – معرفی کد vhdl و تشریح سیر تکمیلی مدار

در این بخش از ما خواسته شده که ALU را ارتقا دهیم تا Shift و Rotation به‌صورت چند بیتی انجام شود. در ادامه به توضیح هر کدام از ماژول‌های به‌روزرسانی شده می‌پردازیم.

- **Variable Rotation Left without Carry & with Carry**

در این Component با سیگنال Op بین دو عملگر یکی را انتخاب می‌کنیم؛ اگر Op=0 باشد، عمل Rotation بدون احتساب Carry انجام شده و مقدار Cout برابر صفر می‌شود. در غیر این‌صورت Rotation با Carry انجام می‌شود. در این عملگر دو ورودی ۸ بیتی به نام‌های Input1 و Input2 داریم که عمل Rotation بر ورودی اعمال می‌شود و ورودی دوم تعداد دفعات اعمال عملیات را نشان می‌دهد. در Rotation بدون Carry، پس از ۸ بار Rotate مقدار ورودی اولیه و خروجی نهایی برابر می‌شود؛ پس به همین دلیل باقی‌ماندهٔ تقسیم ورودی دوم به ۸ را پیدا کرده و در B می‌ریزیم. این مقدار برای Rotation با Carry برابر ۹ می‌شود؛ بنابراین باقی‌مانده ورودی دوم را به ۹ محسابه کرده و در C می‌ریزیم.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Rotation IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cin : IN STD_LOGIC;
        Op : IN STD_LOGIC;
        Output1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC);
END Rotation;

ARCHITECTURE Behavioral OF Rotation IS

    -- Input1 with Carry
    SIGNAL Input1_C : STD_LOGIC_VECTOR (8 DOWNTO 0);

    -- Output1 with Carry
    SIGNAL Output1_C : STD_LOGIC_VECTOR (8 DOWNTO 0);
```

```
BEGIN

    Input1_C <= Cin & Input1;

    PROCESS (Input1, Input2, Op, Input1_C, Output1_C)

       -- NumofPosVRL : Number of Position of Variable Rotation Left
        VARIABLE NumofPosVRL : INTEGER;

       -- NumofPosVRL : Number of Position of Variable Rotation Left with Carry
        VARIABLE NumofPosVRLC : INTEGER;

    BEGIN

       -- Rotation is periodic with period 8.
        NumofPosVRL := CONV_INTEGER(Input2) REM 8;

       -- Rotation with carry is periodic with period 9.
        NumofPosVRLC := CONV_INTEGER(Input2) REM 9;

       -- Variable Rotation Left: Op = '0'
       -- Variable Rotation Left with Carry: Op = '1'

       -- Determining Output1 and Cout for Each Operation
        IF Op = '0' THEN

            IF NumofPosVRL = 0 THEN
                Output1 <= Input1;

            ELSE
                Output1 <= Input1(7 - NumofPosVRL DOWNTO 0) & Input1(7 DOWNTO 8
 - NumofPosVRL);

            END IF;

            Cout <= '0';
            Output1_C <= (OTHERS => '0');

        ELSE

            IF NumofPosVRLC = 0 THEN
                Output1_C <= Input1_C;
```

```
            ELSE
                Output1_C <= Input1_C(8 - NumofPosVRLC DOWNTO 0) & Input1_C(8 D
OWNTO 9 - NumofPosVRLC);

            END IF;

            Output1 <= Output1_C(7 DOWNTO 0);
            Cout <= Output1_C(8);

        END IF;

    END PROCESS;

END Behavioral;
```

کد شماره ۲۴- نحوه عملکرد *Variable Rotation Left* و تعیین خروجی‌ها

- **Variable Logical & Arithmetic Shift Right**

همانند بخش قبل، با توجه به مقدار Op، یکی از عملگرها را انتخاب می‌کنیم. اگر Op=0 باشد، به صورت Logical Shift Right انجام می‌شود؛ در غیر این صورت این عملیات به صورت **Arithmetic** اجرا می‌شود. در این بخش نیز دو ورودی Input1 و Input2 تعریف می‌شود تا عملیاتی بر آن اعمال شود. برای این‌کار، ورودی دوم را به‌صورت Integer در سیگنال B می‌ریزیم؛ اگر مقدار B برابر ۰ باشد، ورودی بدون تغییر به خروجی فرستاده می‌شود و مقدار Cout، برابر ۰ می‌شود. همچنین اگر $1 \leq B \leq 7$ باشد، ورودی به‌اندازه B واحد به سمت راست Shift می‌خورد و باقی بیت‌ها از سمت راست یکی یکی حذف می‌شود تا به همان تعداد، ۰ جایگزین بیت‌های سمت چپ ورودی شود. مقدار Cout در این مرحله، برابر آخرین بیت حذف شدهٔ ورودی است. اگر $B = 8$ شود، تمام بیت‌های ورودی صفر شده و مقدار Cout نیز برابر بیت سمت چپ ورودی می‌شود. در دیگر حالات تمام بیت‌های ورودی و نیز Cout برابر صفر می‌شوند.

همانطور که اشاره شد، اگر Op=1 باشد، Shift به‌صورت Arithmetic انجام می‌شوند. تفاوت این حالت با حالت قبلی در آن است که به‌جای صفرهای تولید شده در سمت چپ ورودی، این‌بار بیت‌های تولید شده برابر با بیت سمت چپ ورودی می‌باشد؛ همچنین در پایان، مقدار بدست آمده با بیت شماره B-1 اُم ورودی جمع می‌شود. تمام حالت‌های مختلف B و تاثیر آن بر خروجی در این بخش نیز صادق است.

```vhdl
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Shift_Right IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Op : IN STD_LOGIC;
        Output1 : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC;
        N : OUT STD_LOGIC);
END Shift_Right;

ARCHITECTURE Behavioral OF Shift_Right IS

BEGIN

    -- Variable Logic Shift Right : Op = '0'
    -- Variable Arithmetic Shift Right with Rounding : Op = '1'
    -- Determining N for each Operation
    N <= Output1(7) WHEN Op = '1' ELSE
        '0';

    PROCESS (Input1, Input2, Op)

        -- NumofPos : Number of Position
        VARIABLE NumofPos : INTEGER;

        -- ZEROS : A Vector of Zeros
        VARIABLE ZEROS : STD_LOGIC_VECTOR(7 DOWNTO 0);

        -- S : A Vector of Sign Bit
        VARIABLE S : STD_LOGIC_VECTOR(7 DOWNTO 0);

    BEGIN

        NumofPos := CONV_INTEGER(Input2);

        ZEROS := x"00";

        S := (OTHERS => Input1(7));
```

```
        -- Determining Output1 and Cout for Each Operation
        IF Op = '0' THEN

            IF NumofPos = 0 THEN
                Output1 <= Input1;
                Cout <= '0';

            ELSIF NumofPos >= 1 AND 7 >= NumofPos THEN
                Output1 <= ZEROS(NumofPos - 1 DOWNTO 0)
                           & Input1(7 DOWNTO NumofPos);
                Cout <= Input1(NumofPos - 1);

            ELSIF NumofPos = 8 THEN
                Output1 <= ZEROS;
                Cout <= Input1(7);
            ELSE
                Output1 <= ZEROS;
                Cout <= '0';
            END IF;

        ELSE
            IF NumofPos = 0 THEN
                Output1 <= Input1;
                Cout <= '0';

            ELSIF NumofPos >= 1 AND 7 >= NumofPos THEN
                Output1 <= (S(NumofPos - 1 DOWNTO 0)
                           & Input1(7 DOWNTO NumofPos)) + Input1(NumofPos - 1);
                Cout <= Input1(NumofPos - 1);

            ELSIF NumofPos = 8 THEN
                Output1 <= S + Input1(7);
                Cout <= Input1(7);
            ELSE
                Output1 <= S;
                Cout <= '0';

            END IF;
        END IF;
    END PROCESS;

END Behavioral;
```

کد شماره ۲۵- نحوه عملکرد Variable Logical & Arithmetic Shift Right و تعیین خروجی‌ها

- Variable Logical Shift Left

مطابق دو بخش قبل، دو ورودی Input1 و Input2 تعریف می‌شود تا ورودی دوم نشان‌دهنده مقدار Shift باشد. این ورودی را به‌صورت Integer در سیگنال B می‌ریزیم. اگر مقدار B صفر شود، ورودی بدون تغییر به خروجی فرستاده می‌شود و مقدار Cout هم برابر صفر خواهد بود. حال اگر $1 \leq B \leq 7$ باشد، ورودی به اندازه B به سمت چپ Shift می‌خورد. باقی بیت‌ها از سمت چپ یکی یکی حذف می‌شود و به همان تعداد، بیت‌های سمت راست ورودی صفر می‌شود. مقدار Cout در این مرحله، برابر آخرین بیت حذف شدهٔ ورودی است. اگر B=8 اتخاذ شود، تمام بیت‌های ورودی صفر شده و مقدار Cout نیز برابر بیت سمت راست ورودی می‌شود. در دیگر حالات تمام بیت‌های ورودی و نیز Cout، برابر صفر است.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Shift_Left IS
    PORT (
        Input1, Input2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        Output1 : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        Cout : OUT STD_LOGIC;
        N : OUT STD_LOGIC);
END Shift_Left;

ARCHITECTURE Behavioral OF Shift_Left IS

BEGIN

    -- Determining N
    N <= Output1(7);

    PROCESS (Input1, Input2)

        -- NumofPos : Number of Position
        VARIABLE NumofPos : INTEGER;

        -- ZEROS : A Vector of Zeros
        VARIABLE ZEROS : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
    BEGIN

        NumofPos := CONV_INTEGER(Input2);
        ZEROS := x"00";
        IF NumofPos = 0 THEN
            Output1 <= Input1;
            Cout <= '0';

        ELSIF NumofPos >= 1 AND NumofPos <= 7 THEN
            Output1 <= Input1(7 - NumofPos DOWNTO 0) & ZEROS(NumofPos - 1 DOWNT
O 0);
            Cout <= Input1(8 - NumofPos);

        ELSIF NumofPos = 8 THEN
            Output1 <= x"00";
            Cout <= Input1(0);

        ELSE
            Output1 <= x"00";
            Cout <= '0';

        END IF;

    END PROCESS;

END Behavioral;
```

کد شماره ۲۶- نحوه عملکرد Variable Logical Shift Left و تعیین خروجی‌ها

۳ – ۲ – بررسی آزمایش test bench و اعتبارسنجی کد vhdl

فرم کلی نحوه تعریف تست بنچ و بهره‌برداری از آن، مفصلا در بخش ۱-۲ تشریح شد. اکنون توجه شما را به بخشی از کد تست‌بنچ در نظر گرفته شده برای بخش امتیازی پروژه یا "Bonus Task" جلب می‌کنیم: (همانطور که در بخش ۱-۲ نیز اشاره کردیم، برای پرهیز از شلوغی، از ارائه کامل کد در گزارش خودداری کرده‌ایم. خواهشمند است که برای مشاهده کامل کد، به فایل tb_ALU.vhd مربوط به این بخش مراجعه فرمایید.)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY tb_ALU IS
END tb_ALU;

ARCHITECTURE behavior OF tb_ALU IS
    COMPONENT ALU
        PORT (
            A : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            B : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            Cin : IN STD_LOGIC;
            OPCODE : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            X : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            Y : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0);
            Z : INOUT STD_LOGIC;
            Cout : INOUT STD_LOGIC;
            V : INOUT STD_LOGIC;
            F_active : OUT STD_LOGIC;
            X_bin_pal : OUT STD_LOGIC;
            X_prime : OUT STD_LOGIC;
            N : OUT STD_LOGIC
        );
    END COMPONENT;
```

```
    -- Defining Signals
    SIGNAL A : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
    SIGNAL B : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
    SIGNAL Cin : STD_LOGIC := '0';
    SIGNAL OPCODE : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
    SIGNAL Z : STD_LOGIC;
    SIGNAL Cout : STD_LOGIC;
    SIGNAL V : STD_LOGIC;
    SIGNAL X : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL Y : STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL F_active : STD_LOGIC;
    SIGNAL X_bin_pal : STD_LOGIC;
    SIGNAL X_prime : STD_LOGIC;
    SIGNAL N : STD_LOGIC;

BEGIN

    UUT : ALU PORT MAP(
        A => A,
        B => B,
        Cin => Cin,
        OPCODE => OPCODE,
        X => X,
        Y => Y,
        Z => Z,
        Cout => Cout,
        V => V,
        F_active => F_active,
        X_bin_pal => X_bin_pal,
        X_prime => X_prime,
        N => N
    );

    -- Stimuluation Process
    stim_proc : PROCESS

    BEGIN

        -- Determining Signal A
        A <= x"ac";
```

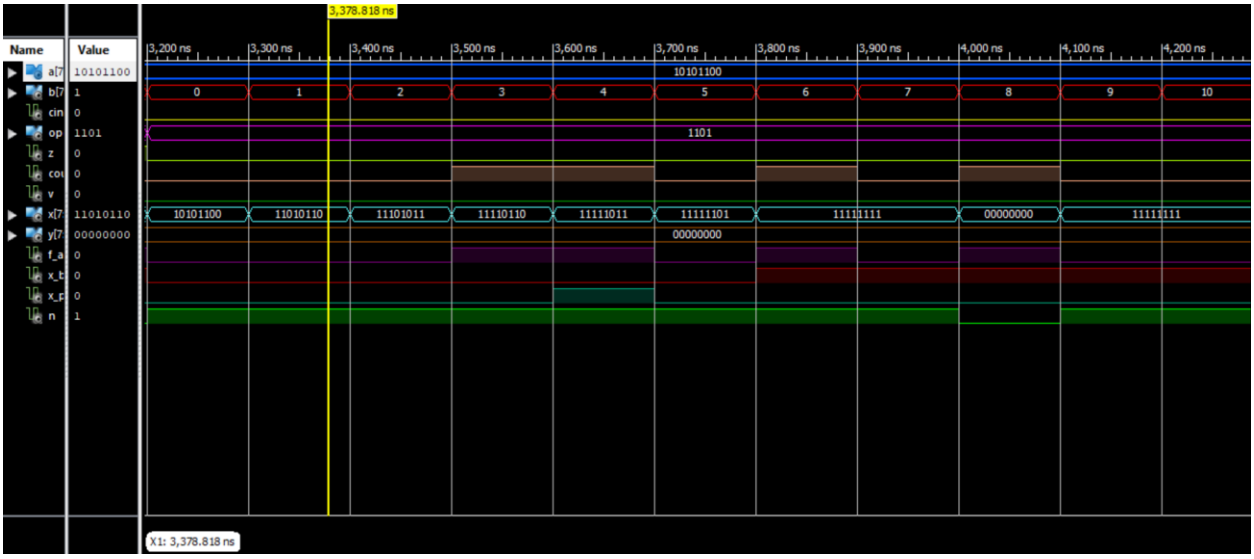
```
        -------------------------- Variable Rotation Left
        OPCODE <= x"a";
        Cin <= '0';

        B <= x"00";
        WAIT FOR 100 ns;
        B <= x"01";
        WAIT FOR 100 ns;
        B <= x"02";
        WAIT FOR 100 ns;
        B <= x"03";
        WAIT FOR 100 ns;
        B <= x"04";
        WAIT FOR 100 ns;
        B <= x"05";
        WAIT FOR 100 ns;
        B <= x"06";
        WAIT FOR 100 ns;
        B <= x"07";
        WAIT FOR 100 ns;
        B <= x"08";
        WAIT FOR 100 ns;
        B <= x"09";
        WAIT FOR 100 ns;

       .
       .
       .

        -------------------------
- Variable Arithmetic Shift Right with Rounding - Exceptional Cases
        OPCODE <= x"d";

        A <= x"ff";
        B <= x"01";
        WAIT FOR 100 ns;

        B <= x"02";
        WAIT FOR 100 ns;

        A <= x"fe";
        B <= x"01";
        WAIT FOR 100 ns;
```

```
        B <= x"02";
        WAIT FOR 100 ns;

        B <= x"03";
        WAIT FOR 100 ns;

        WAIT;

    END PROCESS;

END;
```

کد شماره ۲۷- *Test Bench-Variable Arithmetic Shift Right*

اکنون یک مورد از نتایج آزمایش تست‌بنچ را بررسی می‌کنیم. مطابق بخش ۱-۲ و ۲-۲ ، برای پرهیز از شلوغی، از ارائه تمام نتایج تست‌بنچ در گزارش خودداری می‌کنیم. خواهشمند است که نتایج باقی آزمایش‌ها را که به صورت فایل تصویری در پوشه BonusTaskTestBenchSCs قرار داده‌ایم، مشاهده فرمایید. ترتیب نام‌گذاری نتایج آزمایش تست‌بنچ در این پوشه، مطابق مقدار operation code و بازه زمانی آزمایش برای هر عملیات است.

تصمیم داریم نتایج آزمایش، به ازای عملگر Variable Arithmetic Shift Right with Rounding را بررسی کنیم؛ دلیل این انتخاب، فرم خاص و پیچیدگی بیشتر این عملگر نسبت به سایر عملگرها است. به هر حال تلاش می‌کنیم تا موضوع را روشن کینم. نتیجه آزمایش، با توجه به مقدار ورودی‌ها و مقدار Operation Code، به فرم زیر است:



شکل ۴-نتایج Test Bench-Arithmetic Shift Right

به عنوان مثال، خروجی را به ازای B = 5 ، تحلیل می‌کنیم:

$$A = (1010\ 1100)_2 \rightarrow Arithmetic\ Shift\ to\ right\ by\ 5 : A_{new} = (1111\ 1101)_2$$

از طرفی:

$$Cout = A(B-1) = A(4) = 0 \rightarrow R = A_{new} + 0 = A_{new} = (1111\ 1101)_2$$

یا به ازای B = 3 داریم:

$$A = (1010\ 1100)_2 \rightarrow Arithmetic\ Shift\ to\ right\ by\ 3 : A_{new} = (1111\ 0101)_2$$

از طرفی:

$$Cout = A(B-1) = A(2) = 1 \rightarrow R = A_{new} + 1 = (1111\ 0110)_2$$

۳ – ۳ – نکات جانبی پیرامون نتایج آزمایش

به عنوان نکات حاشیه‌ای در این بخش، موارد زیر را مطرح می‌کنیم:

اول: اگر به شکل موج Cout در عملگر Arithmetic Shift Right دقت کنید؛ متوجه خواهید شد که اگر شکل موج را به فرم اعداد باینری بنویسیم، برابر Reverse عدد ورودی است. دلیل این امر آن است که هر بار با اعمال شیفت، عدد حذف شده، وارد Cout می‌شود؛ پس Reverse عدد ورودی حاصل می‌شود.

دوم: برای پرهیز از پیچیدگی نتایج آزمایش تست‌بنچ، به جای تغییر ورودی A ، تصمیم گرفتیم که مقدار B را تغییر دهیم تا به نوعی رسالت اصلی مدار را به نمایش گذاشته باشیم.

سوم: شمای کلی مدار ساخته شده، تحت عنوان فایل pdf به نام Schematic Bonus Task به فایل‌های پروژه ضمیمه شده است.

جمع‌بندی و نتیجه‌گیری

در این گزارش تلاش کرده‌ایم تا نحوه طراحی یک واحد محاسبه‌گر منطقی یا "ALU" را تشریح کنیم. در هر بخش، فرم صحیح نوشتار کد vhdl را ارائه کردیم و همچنین نتایج آزمایش تست بنچ برای هر عملگر را شرح داده‌ایم.

آن چه که حائز اهمیت است، طراحی ALU با کمک طراحی Component است. در واقع با توجه به عملگرهای متنوع موجود در این واحد، به این نتیجه رسیدیم که بهترین راه برای طراحی این قطعه، تبدیل مسئله به زیرمسئله‌هایی ساده‌تر و کوچک‌تر است. این نحوه تفکر، رایج‌ترین خط فکری میان مهندسانی است که سعی دارند راه‌حلی برای مشکلات بیابند یا دست به ابداع بزنند. تفکری که در کلاس درس بارها به آن اشاره شد و یادآور نقل‌قولی آشناست: «بچه‌ها! ماژولار فکر کنید!»

لازم به ذکر است که برای طراحی پروژه از نرم‌افزارهای زیر کمک گرفته‌ایم:

- **ISE Design Suite V14.7**
- **Xilinx Vivado Design Suite HLx Edition**

«پایان»