به نام خدا

دانشکده مهندسی کامپیوتر

نام درس:
آزمایشگاه معماری کامپیوتر

استاد:
دکتر پریا دربانی

گزارش پروژه

فاطمه‌زهرا بخشنده (بدون گروه)

آذر 1401

هدف آزمایش:

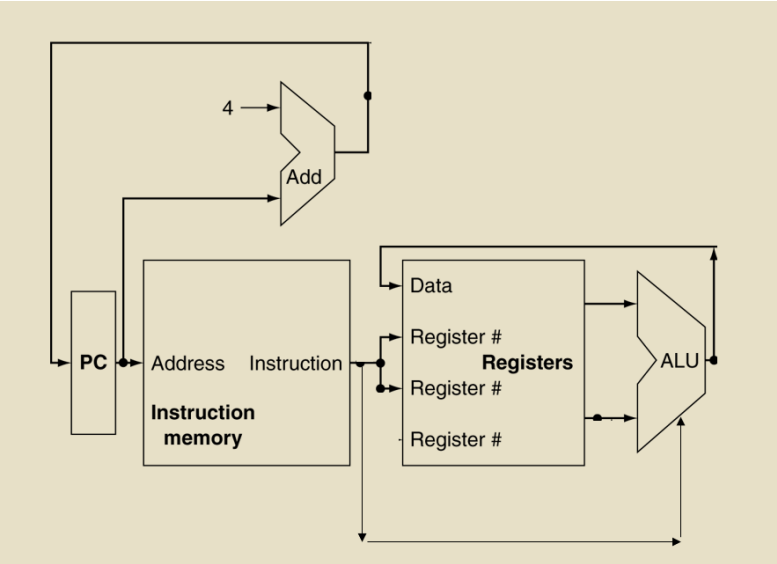
پیاده سازي يک CPU (کامپيوتر) 4 بيتی با استفاده از نرم افزار ISE

نحوه کار:

يك سيستم كامپيوتر پايه از 4 قسمت Control unit , ALU، BUS و ثبات ها تشكيل شده است.

يكي از ثباتهاي كارا در سازمان كامپيوتر، ثبات دستورالعمل است. يك دستورالعمل در 3 مرحله Decode، Fetch و Execute انجام مي شود. به طور خلاصه، كد باينري هر دستورالعمل، در مرحله Fetch از حافظه خوانده و وارد ثبات دستورالعمل مي شود. در مرحله Decode اين كد توسط واحد كنترل كننده ارزيابي مي شود و سيگنال هاي كنترلي لازم براي انجام دستورالعمل ساخته خواهند شد. درنهايت مرحله آخر، عمل منطقي يا رياضي، ورودي و خروجي، انتقالي يا ثباتي مورد نظر دستورالعمل را انجام مي دهد.

ساختار کلی این کامپیوتر به صورت زیر است:



ابتدا بر اساس آدرس موجود در PC یک خط از حافظه خوانده می شود و مقدار PC آپدیت میگردد.

این خط شامل آدرس دو رجیستر (برای خواندن) و آدرس یک رجیستر برای نوشتن نتیجه است. همچنین نوع عملیاتی که قرار است روی آن ها انجام شود در این خط کد وجود دارد.

بر اساس شماره رجیسترها محتوای دو رجیستر خوانده میشود.

عملیات ALU بر اساس کدی که از حافظه خواندیم انجام میشود و نتیجه در رجیستر و در آدرس تعیین شده ذخیره میگردد.

شرح پیاده‌سازی:

ابتدا کامپوننت های مختلف این CPU را پیاده سازی می کنیم.

- کامپوننت ALU:

این کامپوننت دو ورودی 4 بیتی داده دارد و در نهایت حاصل xor این دو ورودی درون خروجی قرار می‌گیرد. این قطعه قسمت منطقی و عملیاتی کامپیوتر است که داده ها را از ثبات ها گرفته و نتیجه xor را بر حسب نیاز به خروجی تحویل میدهد و به Register File می فرستد تا درون ادرس مشخص شده نوشته شود .

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity CPU_ALU is
    Port (
        a : in STD_LOGIC_VECTOR (3 downto 0);
        b : in STD_LOGIC_VECTOR (3 downto 0);
        x : out STD_LOGIC_VECTOR (3 downto 0)
    );
end CPU_ALU;

architecture Behavioral of CPU_ALU is
begin
    x <= a xor b ;
end Behavioral;
```

- کامپوننت Instruction Memory:

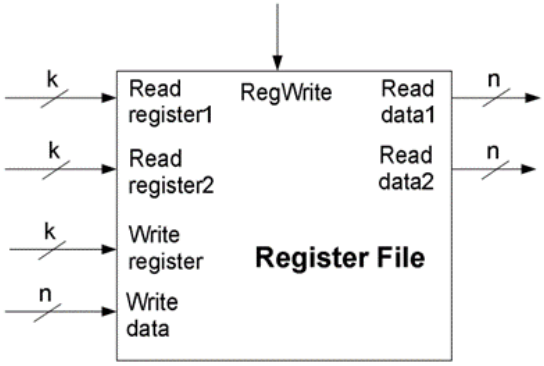
این کامپوننت یک آدرس به عنوان ورودی می گیرد و به این آدرس در memory رفته و مقدار درون آن را به عنوان instruction بر می گرداند. در واقع این مرحله، مرحله Fetch است که کد باینري هر دستورالعمل، از حافظه خوانده و سپس وارد ثبات دستورالعمل مي شود.

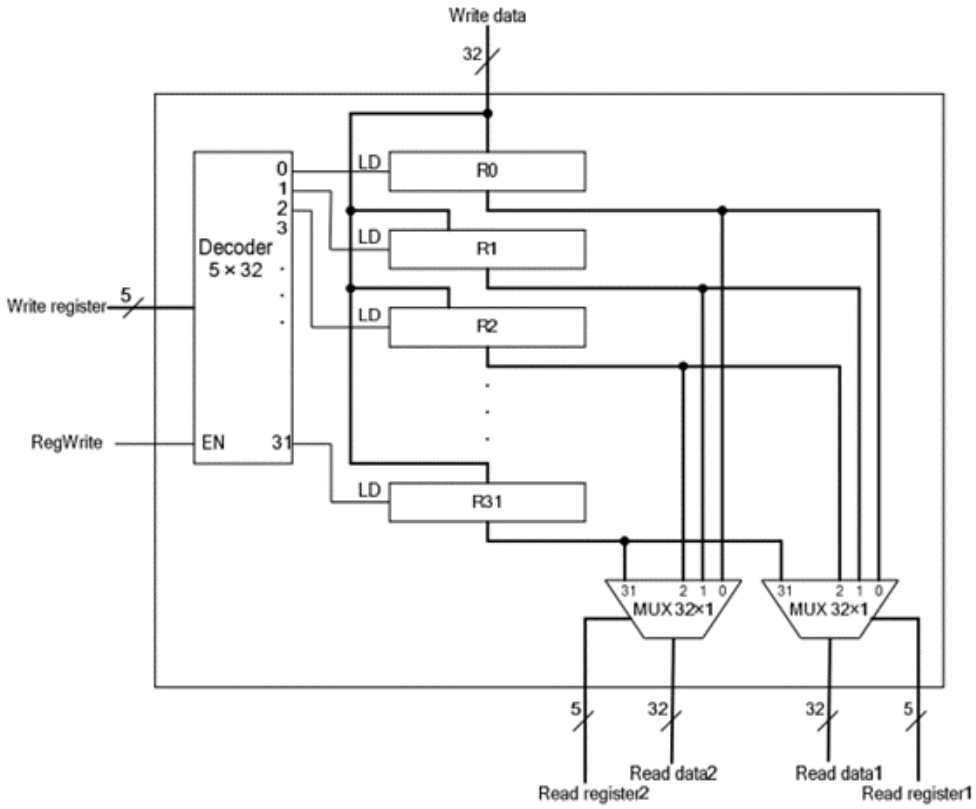
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity CPU_Instruction_Mem is
    Port (
        address : in STD_LOGIC_VECTOR (4 downto 0);
        instruction : out STD_LOGIC_VECTOR (14 downto 0)
    );
end CPU_Instruction_Mem;

architecture Behavioral of CPU_Instruction_Mem is

type mem_block is std_logic_vector(14 downto 0);
type mem_array is array(0 to 15) of mem_block;

signal mem_block_0: mem_block := "101100000110111";
signal mem_block_1: mem_block := "101100000110111";
signal mem_block_2: mem_block := "101100000110111";
signal mem_block_3: mem_block := "101100000110111";
signal mem_block_4: mem_block := "101100000110111";
signal mem_block_5: mem_block := "101100000110111";
signal mem_block_6: mem_block := "101100000110111";
signal mem_block_7: mem_block := "101100000110111";
signal mem_block_8: mem_block := "101100000110111";
signal mem_block_9: mem_block := "101100000110111";
signal mem_block_10: mem_block := "101100000110111";
signal mem_block_11: mem_block := "101100000110111";
signal mem_block_12: mem_block := "101100000110111";
signal mem_block_13: mem_block := "101100000110111";
signal mem_block_14: mem_block := "101100000110111";
```

```
signal memory : mem_array := (
    mem_block_0,
    mem_block_1,
    mem_block_2,
    mem_block_3,
    mem_block_4,
    mem_block_5,
    mem_block_6,
    mem_block_7,
    mem_block_8,
    mem_block_9,
    mem_block_10,
    mem_block_11,
    mem_block_12,
    mem_block_13,
    mem_block_14
);

begin
    instruction <= memory(
        to_integer(unsigned(address))
    );
end Behavioral;
```

❖ کامپوننت Register File:

این کامپوننت با توجه به شکل زیر پیاده سازی شده است.





ورودی های clk، RegWrite، write_data، read_reg1 و read_reg2 را گرفته و read_data1 و read_data2 را خروجی می دهد. در واقع کار خواندن محتوای رجیستر های ورودی و دادن آن ها به ALU، و همچنین نوشتن نتیجه ALU روی رجیستر سوم را انجام می دهد.

32 رجیستر برای این قسمت تعریف شده است.

کد آن را در صفحه های بعدی می بینیم.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity CPU_Register_File is
    Port (
        clk: in std_logic ;
        RegWrite : in STD_LOGIC_VECTOR (4 downto 0);
        write_data : in STD_LOGIC_VECTOR (3 downto 0);
        read_reg1 : in STD_LOGIC_VECTOR (4 downto 0);
        read_data1 : out STD_LOGIC_VECTOR (3 downto 0);
        read_reg2 : in STD_LOGIC_VECTOR (4 downto 0);
        read_data2 : out STD_LOGIC_VECTOR (3 downto 0)
    );
end CPU_Register_File;

architecture Behavioral of CPU_Register_File is

type register_block is std_logic_vector(3 downto 0);
type register is array (0 to 31) of register_block;
```

```
type register_block is std_logic_vector(3 downto 0);
type register is array (0 to 31) of register_block;

signal register_block31: register_block := "00011111"
signal register_block30: register_block := "00011110"
signal register_block29: register_block := "00011101"
signal register_block28: register_block := "00011100"
signal register_block27: register_block := "00011011"
signal register_block26: register_block := "00011010"
signal register_block25: register_block := "00011001"
signal register_block24: register_block := "00011000"
signal register_block23: register_block := "00010111"
signal register_block22: register_block := "00010110"
signal register_block21: register_block := "00010101"
signal register_block20: register_block := "00010100"
signal register_block19: register_block := "00010011"
signal register_block18: register_block := "00010010"
signal register_block17: register_block := "00010001"
signal register_block16: register_block := "00010000"
signal register_block15: register_block := "00001111"
signal register_block14: register_block := "00001110"
signal register_block13: register_block := "00001101"
signal register_block12: register_block := "00001100"
signal register_block11: register_block := "00001011"
signal register_block10: register_block := "00001010"
signal register_block9: register_block := "00001001"
signal register_block8: register_block := "00001000"
signal register_block7: register_block := "00000111"
signal register_block6: register_block := "00000110"
signal register_block5: register_block := "00000101"
signal register_block4: register_block := "00000100"
signal register_block3: register_block := "00000011"
signal register_block2: register_block := "00000010"
signal register_block1: register_block := "00000001"
signal register_block0: register_block := "00000000"
```

```
signal RegisterFile: register := (
    register_block0,
    register_block1,
    register_block2,
    register_block3,
    register_block4,
    register_block5,
    register_block6,
    register_block7,
    register_block8,
    register_block9,
    register_block10,
    register_block11,
    register_block12,
    register_block13,
    register_block14,
    register_block15,
    register_block16,
    register_block17,
    register_block18,
    register_block19,
    register_block20,
    register_block21,
    register_block22,
    register_block23,
    register_block24,
    register_block25,
    register_block26,
    register_block27,
    register_block28,
    register_block29,
    register_block30,
    register_block31,
);
```

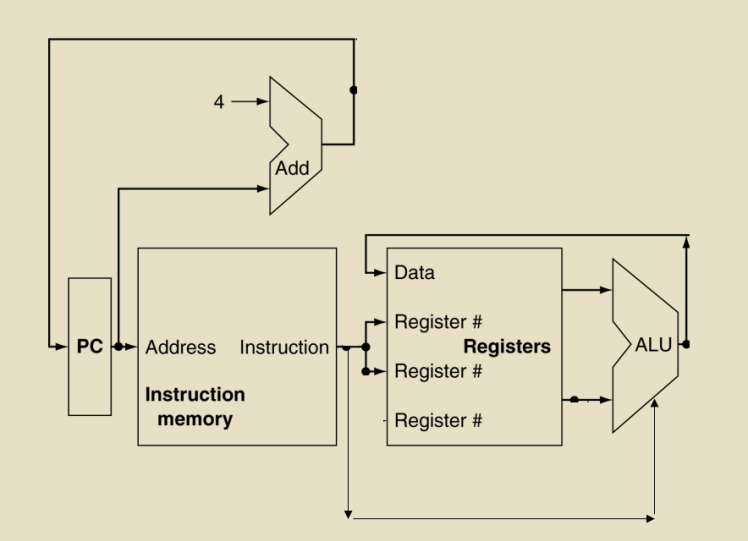
```
begin
    read_data1 <= RegisterFile(
        to_integer(unsigned(read_reg1))
    );

    read_data2 <= RegisterFile(
        to_integer(unsigned(read_reg2))
    );

    process (clk)
    begin

        if rising_edge(clk) then
            RegisterFile(
                to_integer(unsigned(RegWrite))
            ) <= write_data ;
        end if;

    end process;
end Behavioral;
```

❖ کامپوننت Adder:

این کامپوننت نقش جمع کردن PC با عدد 4 (0100) را دارد. نحوه کارکرد آن به تفضیل در آزمایش های 3 و 4 بررسی شد.

❖ کامپوننت CPU:

این کامپوننت، کامپوننت نهایی و اصلی ماست. و از تمام کاممپوننت های پیشین استفاده می‌کند. در واقع یک CPU، به صورت زیر پیاده سازی می شود.



پس کافی است به صورت structural، کامپوننت های پیشین را با توجه به تصویر بالا به هم وصل کنیم. برای این کار همه component ها را در CPU، تعریف کرده و در هنگام استفاده از آن ها، از port map استفاده می کنیم.

می توانیم کد این کامپوننت را مشاهده کنیم.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity CPU_4Bit is
end CPU_4Bit;
```

```
architecture Behavioral of CPU_4Bit is

    Component CPU_ALU is
    Port (
        a : in STD_LOGIC_VECTOR (3 downto 0);
        b : in STD_LOGIC_VECTOR (3 downto 0);
        x : out STD_LOGIC_VECTOR (3 downto 0)
    );
    end Component;

    Component FullAdder4Bit is
    Port (
        clk : in  STD_LOGIC;
        A : in  STD_LOGIC_Vector(3 downto 0);
        B : in  STD_LOGIC_Vector(3 downto 0);
        Cin : in  STD_LOGIC;
        Cout : out  STD_LOGIC;
        S : out  STD_LOGIC_Vector(3 downto 0)
    );
    end Component;

    Component CPU_Instruction_Mem is
        Port (
            address : in STD_LOGIC_VECTOR (4 downto 0);
            instruction : out STD_LOGIC_VECTOR (14 downto 0)
        );
    end Component;
```

```
    Component CPU_Register_File is
        Port (
            clk: in std_logic;
            RegWrite : in STD_LOGIC_VECTOR (4 downto 0);
            write_data : in STD_LOGIC_VECTOR (3 downto 0));
            read_reg1 : in STD_LOGIC_VECTOR (4 downto 0);
            read_data1 : out STD_LOGIC_VECTOR (3 downto 0);
            read_reg2 : in STD_LOGIC_VECTOR (4 downto 0);
            read_data2 : out STD_LOGIC_VECTOR (3 downto 0)
        );
    end Component;


    signal clk: std_logic := '0';
    Signal PC: integer := 0;
    signal instruction : std_logic_vector(14 downto 0);
    signal read_reg1, read_reg2, RegWrite : std_logic_vector(4 downto 0);
    signal read_data1, read_data2, write_data: std_logic_vector(3 downto 0);
```

```
begin

    I_M: CPU_Instruction_Mem port map(
        address <= std_logic_vector(to_unsigned(PC, 5)),
        instruction <= instruction
    );

    -- split
    read_reg1 <= instruction(14 downto 10) ;
    read_reg2 <= instruction(9 downto 5) ;
    RegWrite <= instruction(4 downto 0);
```

```
R_F: CPU_Register_File port map(
    clk => clk
    RegWrite => RegWrite,
    write_data => write_data,
    read_reg1 => read_reg1,
    read_data1 => read_data1,
    read_reg2 => read_reg2,
    read_data2= > read_data2,
);

A_L: CPU_ALU port map(
    a => read_data1,
    b => read_data2,
    c => write_data
);
```

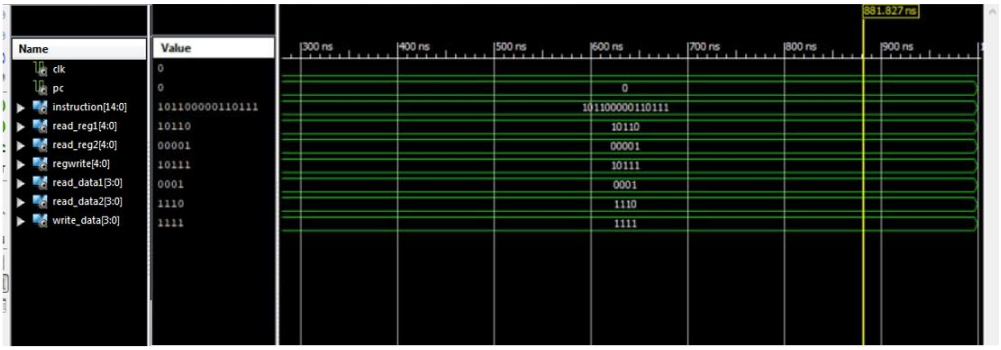
```
    process (clk)
    begin
        if rising_edge(clk) then
            if PC == 0 then
                F1: FullAdder4Bit Port map(
                    clk <= clk,
                    A <= to_unsigned(PC, 5),
                    B <= '00011',
                    Cin <= '0',
                    Cout <= None,
                    S <= PC
                );
                --PC <= PC + 3;

            elsif PC < 15 then
                F1: FullAdder4Bit Port map(
                    clk <= clk,
                    A <= to_unsigned(PC % 15, 5),
                    B <= '00011',
                    Cin <= '0',
                    Cout <= None,
                    S <= PC
                );
                --PC <= PC + 4;
            else
                PC <= 0;
            end if;
        end if;
    end process;
    clock_process: process
    begin
        clk <= 0;
        wait for 5 ns;
        clk <= 0;
        wait for 5 ns;
        wait;
    end process;
end Behavioral;
```

کد را سنتز می کنیم.

```
=========================================================

Process "Synthesize - XST" completed successfully
```

تست و آزمایش:

همه موارد را در CPU انجام دادیم. (مراحل Fetch و Decode و Execute با توجه به مقدار PC) و حالا موج خروجی آن را رسم می کنیم.



نتیجه گیری:

همانطور که انتظار داشتیم دقیقا رفتار مورد نظر مشاهده می شود. یعنی instruction مورد نظر Fetch و Decode شده و دیتا ها از رجیستر ها خوانده می شوند. و حاصل xor دو دیتا یعنی حاصل xor عدد 1110 و 0001 برابر 1111 شده و در رجیستر مقصد ریخته شده است.