● به نام خدا ●

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده مهندسی برق
دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

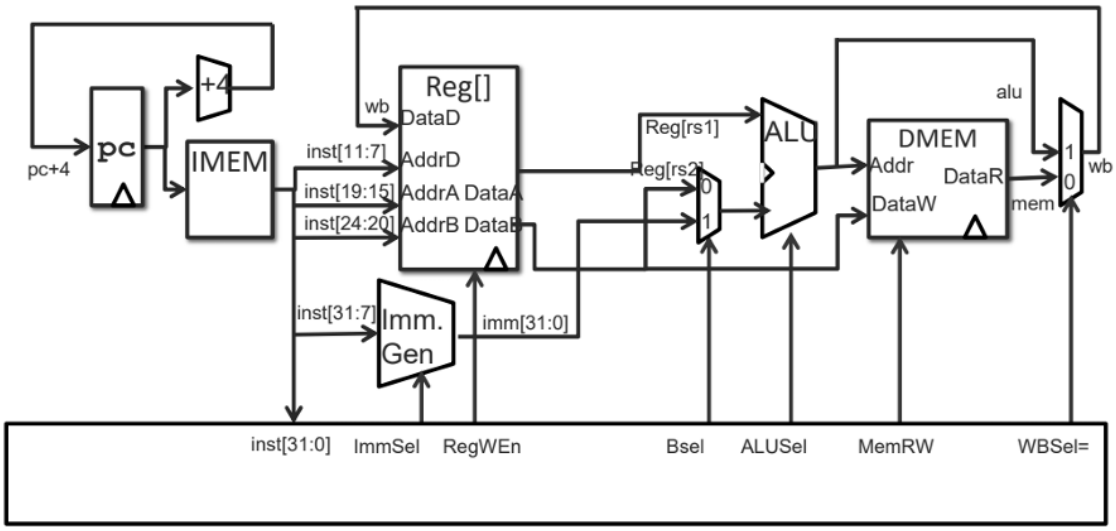
گزارشکار پروژه RISC_V

سیداحمد حسینی میانگله

۴۰۰۲۳۰۲۳

دکتر شعبانی

فهرست



شکل 1 : طراحی انجام شده در این پروژه

مقدمه (Introduction)

در معماری‌های مدرن رایانه، طراحی و پیاده‌سازی قسمت‌های مختلف یک پردازنده نقش بسیار مهمی در عملکرد و قابلیت توسعه آن ایفا می‌کند. به عنوان اولین گام در پروژه پیاده‌سازی یک پردازنده تک سیکل RISC-V، اجزای اصلی این معماری شامل **واحد محاسبه و منطق (ALU)**، **Register File**، **شمارنده برنامه (Program Counter)**، **Immediate Generator**، **واحد کنترل (Control Unit)** و **حافظه دستورالعمل (Instruction Memory)** و .... ، به صورت مجزا طراحی و پیاده‌سازی شده‌اند.

هر یک از این کامپوننت‌ها وظیفه خاصی را بر عهده دارند که اجرای درست و هماهنگ آنها، بسترساز عملکرد صحیح کل پردازنده خواهد بود. ALU عملیات اصلی محاسباتی و منطقی را انجام می‌دهد و با دریافت دو ورودی ۳۲ بیتی و انتخاب عملکرد مورد نیاز بر اساس سیگنال ALU_Sel، نتیجه عملیات‌هایی مانند جمع، تفریق، AND و OR را ارائه می‌کند. رجیستر فایل به‌عنوان بانک ثبات‌های پردازنده، امکان ذخیره و بازیابی سریع داده‌ها را با پشتیبانی از ۳۲ ثبات ۳۲ بیتی فراهم می‌آورد. حافظه دستورالعمل نیز محلی برای نگهداری و بازیابی دستورهای اجرایی در هر سیکل است که با ساختار Byte Addressed پیاده‌سازی شده و در پروژه حاضر تا ۱۶ دستور ذخیره می‌نماید.

شمارنده برنامه (Program Counter) جهت مدیریت توالی دستورهای اجرایی به کار می‌رود و تطبیق کامل با نیاز آدرس‌دهی شش بیتی پردازنده دارد. Immediate Generator نیز امکان گسترش و تبدیل فوریه‌های کوچک‌تر به سیگنال‌های ۳۲ بیتی را برای انواع دستوراتی که نیازمند مقدار فوری هستند، فراهم می‌سازد. واحد کنترل یا Control Unit در این ساختار وظیفه کدگشایی دستورالعمل و تولید سیگنال‌های کنترلی لازم جهت هدایت صحیح داده‌ها در مسیر داده پردازنده را برعهده دارد، به گونه‌ای که تمام عملیات R-type مانند AND، OR، ADD و SUB و همچنین دستورهای مهم I-type مانند ADDI، ANDI و ORI پشتیبانی شوند.

در ادامه، ساختار، ویژگی‌ها و شیوه پیاده‌سازی هر یک از این ماژول‌های کلیدی به صورت کامل و همراه با کد VHDL ارائه می‌گردد تا مراحل تحقق سخت‌افزاری هر بخش از پردازنده به شکل شفاف و مستند قابل بررسی باشد.

شرح اجزای پیاده‌سازی (Module Description)

شمارنده برنامه (Program Counter)

شمارنده برنامه (Program Counter یا به اختصار PC) یکی از اجزای اصلی در معماری پردازنده است که وظیفه نگهداری آدرس دستور فعلی را بر عهده دارد و با هر سیکل کلاک به آدرس دستور بعدی به‌روزرسانی می‌شود. در معماری RISC-V تک‌چرخه، اندازه آدرس PC در این پروژه ۶ بیت (تا ۶۴ خانه دستورالعمل) در نظر گرفته شده است.

ورودی‌ها شامل سیگنال کلاک (Clk)، سیگنال Reset (ریست غیرهمزمان و فعال با '۱') و مقدار ورودی بعدی PC (PC_in) می‌باشد.

هنگامی که لبه بالارونده کلاک رخ می‌دهد:

اگر سیگنال Reset فعال باشد، مقدار PC به صفر (آدرس شروع برنامه) بازمی‌گردد.

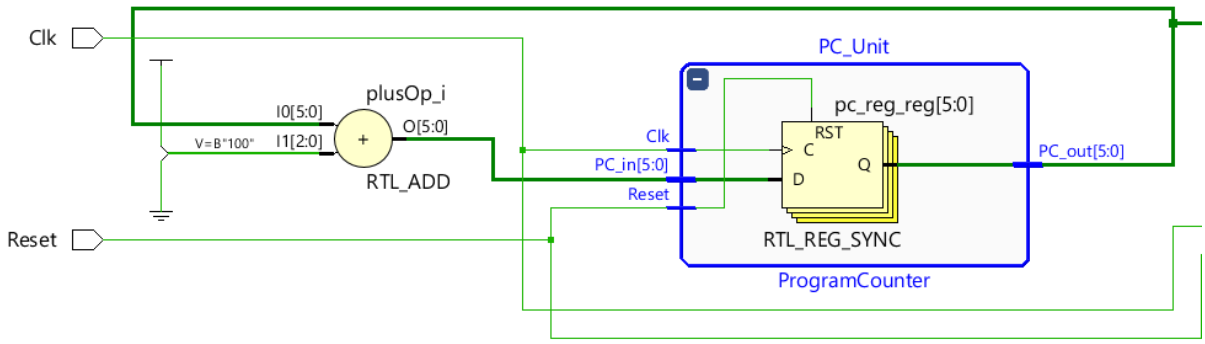
در غیر این صورت، مقدار جدید ورودی (PC_in) در رجیستر داخلی ذخیره می‌شود.

مقدار فعلی PC از طریق پورت خروجی (PC_out) در اختیار سایر بخش‌های پردازنده (مانند Instruction Memory) قرار می‌گیرد.

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/ProgramCounter.vhd

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

-- Entity declaration for Program Counter (PC)
entity ProgramCounter is
    Port (
        Clk    : in  std_logic;                     -- Clock signal
        Reset  : in  std_logic;                     -- Asynchronous reset signal (active high)
        PC_in  : in  std_logic_vector(5 downto 0);  -- Input: Next PC value
        PC_out : out std_logic_vector(5 downto 0)   -- Output: Current PC value
    );
end ProgramCounter;

architecture Behavioral of ProgramCounter is
    signal pc_reg : std_logic_vector(5 downto 0) := (others => '0');  -- Internal register to store PC
begin
    -- Main process: updates PC on each clock rising edge
    process(Clk)
    begin
        if rising_edge(Clk) then
            if Reset='1' then
                pc_reg <= (others => '0');      -- On reset, set PC to zero
            else
                pc_reg <= PC_in;                -- Otherwise, update PC with input value
            end if;
        end if;
    end process;
    PC_out <= pc_reg;  -- Assign internal PC register to output port
end Behavioral;

```

آدرس شمارنده درون رجیستر 6 بیتی (pc_reg) نگهداری می‌شود و خروجی PC همیشه مقدار آخرین رجیستر است. این طراحی ساده، علاوه بر اجرای دستورات پیوسته، امکان بازگشت به ابتدای برنامه در زمان راه‌اندازی یا ریست را فراهم می‌کند. به دلیل یک سیکل بودن معماری، به محض ارائه مقدار جدید به ورودی، مقدار خروجی نیز به‌روزرسانی می‌شود و حافظه دستور می‌تواند دستور جدید را فراخوانی کند.



حافظه دستورالعمل (Instruction Memory)

ماژول "Instruction Memory" وظیفه ذخیره و تحویل دستورالعمل‌های برنامه را بر عهده دارد و یکی از اجزاء کلیدی پردازنده تک‌چرخه RISC-V محسوب می‌شود. این ماژول به‌صورت یک حافظه فقط‌خواندنی (ROM)۶۴ بایتی پیاده‌سازی شده که هر خانه، یک بایت (۸ بیت) را نگه می‌دارد. هر دستور ۳۲ بیتی (۴ بایت) از چهار خانه متوالی خوانده می‌شود. آدرس‌دهی این حافظه بر اساس **بایت** انجام می‌شود، بنابراین PC و ورودی آدرس ۶ بیتی است.

ورودی:

Addr : آدرس بایت (۶ بیت)، محل خواندن دستور فعلی.

خروجی:

InstOut : داده ۳۲ بیتی (دستور کامل).

در هر سیکل، چهار بایت متوالی از آرایه حافظه خوانده شده و با هم ترکیب می‌شوند تا یک دستور کامل ۳۲ بیتی تولید شود . هر چهار بایت به صورت **Little Endian** کنار هم چیده می‌شوند (ابتدا کم ارزش‌ترین بایت، سپس سایر بایت‌ها از آدرس بعدی). ترکیب این چهار بایت، دستور معادل با استاندارد ماشین RISC-V را تولید می‌کند. در نسخه فعلی پروژه، دستورات زیر به صورت hard-code در حافظه قرار گرفته‌اند

(برای تست):

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/InstructionMemory.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity InstructionMemory is
6      Port (
7          Addr    : in  STD_LOGIC_VECTOR (5 downto 0); -- Byte Address
8          InstOut : out STD_LOGIC_VECTOR (31 downto 0)
9      );
10 end InstructionMemory;
11
12 architecture Behavioral of InstructionMemory is
13     -- Define memory array: 64 bytes of 8-bit width (byte-addressable)
14     type mem_array is array (0 to 63) of STD_LOGIC_VECTOR(7 downto 0);
15
16
17 --    signal imem : mem_array := (
18 --      -- addi x1, x0, 5      : 0x00500093
19 --      0 => X"93", 1 => X"00", 2 => X"50", 3 => X"00",
20 --      -- addi x2, x0, 10     : 0x00A00113
21 --      4 => X"13", 5 => X"01", 6 => X"A0", 7 => X"00",
22 --      -- addi x3, x0, 7      : 0x00700193
23 --      8 => X"93", 9 => X"01", 10 => X"70", 11 => X"00",
24 --      -- add  x4, x1, x2     : 0x00208233
25 --      12 => X"33", 13 => X"82", 14 => X"20", 15 => X"00",
26 --      -- addi x0, x0, 0 (NOP): 0x00000013
27 --      16 => X"13", 17 => X"00", 18 => X"00", 19 => X"00",
28 --      -- add  x5, x4, x3     : 0x003202B3
29 --      20 => X"B3", 21 => X"02", 22 => X"32", 23 => X"00",
30 --      others => (others => '0')
31 --    );
```

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/InstructionMemory.vhd

```
--    );



--    signal imem : mem_array := (
--        -- addi x1, x0, 12   : 0x00C00093
--        0 => X"93", 1 => X"00", 2 => X"C0", 3 => X"00",
--        -- addi x2, x0, 10   : 0x00A00113
--        4 => X"13", 5 => X"01", 6 => X"A0", 7 => X"00",
--        -- and x3, x1, x2    : 0x0020F1B3  <-- ****** **00
--        8 => X"B3", 9 => X"F1", 10 => X"20", 11 => X"00",
--        -- or  x4, x1, x2    : 0x0020E233  <-- ****** **00
--        12 => X"33", 13 => X"E2", 14 => X"20", 15 => X"00",
--        others => (others => '0')
--    );



--    signal imem : mem_array := (
--        -- addi x1, x0, 20
--        0 => X"93", 1 => X"00", 2 => X"40", 3 => X"01",
--        -- addi x2, x1, -5
--        4 => X"13", 5 => X"81", 6 => X"B0", 7 => X"FF",
--        -- andi x3, x1, 0x0F
--        8 => X"93", 9 => X"F1", 10 => X"F0", 11 => X"00",
--        -- ori x4, x2, 0x01
--        12 => X"13", 13 => X"62", 14 => X"11", 15 => X"00",
--        others => (others => '0')
--    );
```

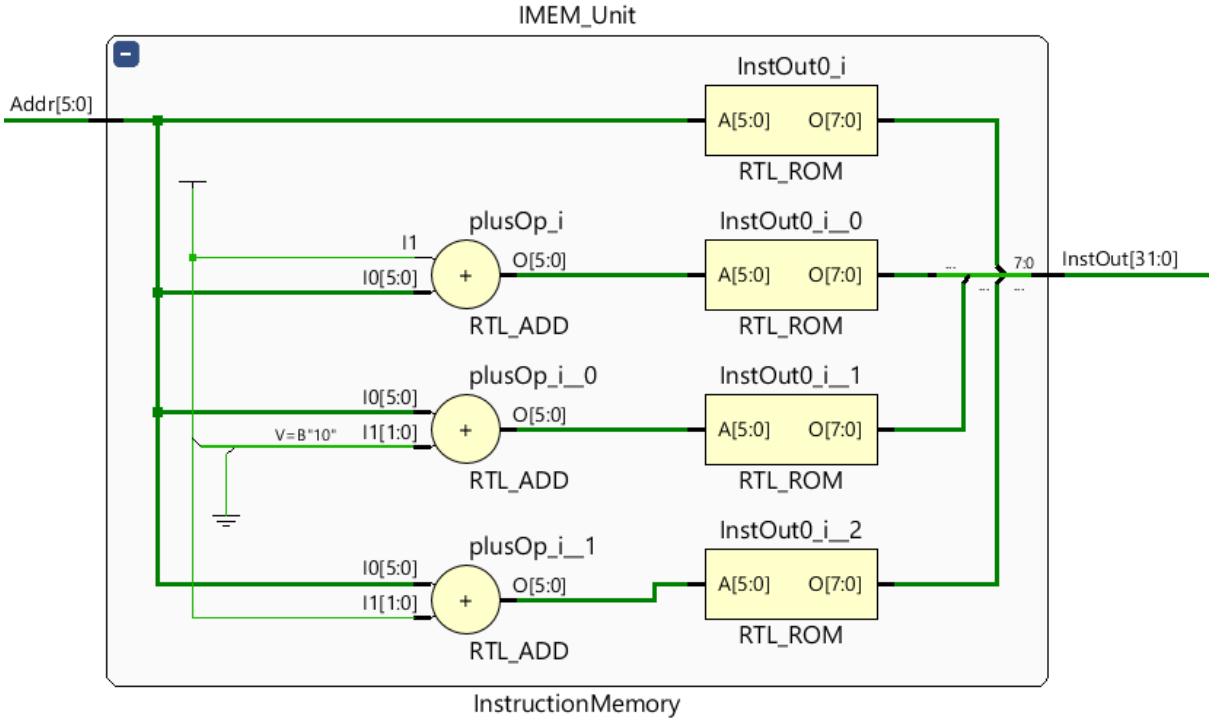
C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/InstructionMemory.vhd

```
--        -- andi x3, x1, 0x0F
--        8 => X"93", 9 => X"F1", 10 => X"F0", 11 => X"00",
--        -- ori x4, x2, 0x01
--        12 => X"13", 13 => X"62", 14 => X"11", 15 => X"00",
--        others => (others => '0')
--    );



    signal imem : mem_array := (
    -- addi x1, x0, 8       : 0x00800093
    0 => x"93", 1 => x"00", 2 => x"80", 3 => x"00",
    -- addi x2, x0, 42      : 0x02A00113
    4 => x"13", 5 => x"01", 6 => x"A0", 7 => x"02",
    -- sw   x2, 0(x1)       : 0x0020A023
    8 => x"23", 9 => x"A0", 10 => x"20", 11 => x"00",
    -- addi x1, x1, -3      : 0xFFD08093
    12 => x"93", 13 => x"80", 14 => x"D0", 15 => x"FF",
    -- lw   x3, 3(x1)       : 0x0030A183
    16 => x"83", 17 => x"A1", 18 => x"30", 19 => x"00",
    others => (others => '0')
    );

begin
    -- Output instruction by combining 4 bytes (little-endian order)
    InstOut <= imem(to_integer(unsigned(Addr))+3)) &    -- Most significant byte
               imem(to_integer(unsigned(Addr))+2)) &
               imem(to_integer(unsigned(Addr))+1)) &
               imem(to_integer(unsigned(Addr)));       -- Least significant byte
end Behavioral;
```

بایت‌های حافظه در آرایه ۶۴ بایتی تعریف شده‌اند و مقادیر اولیه آن‌ها برای شبیه‌سازی بارگذاری شده است.

روش آدرس‌دهی کاملاً **byte-oriented** است؛ هر دستور از آدرس PC، چهار بایت پیوسته را ترکیب

می‌کند تا با استاندارد RISC-V مطابقت داشته باشد. چیدمان چهار بایت، الزامات endian-بودن RISC-V را رعایت می‌کند (ابتدا کم ارزش‌ترین بایت). به‌سادگی می‌توان این آرایه را برای تست سناریوهای مختلف یا دستورالعمل‌های متنوع توسعه داد. مقادیر باقی‌مانده در حافظه صفرگذاری شده‌اند تا از اجرای دستور نامعتبر جلوگیری شود.



واحد تولید مقدار فوری (Immediate Generator)

واحد Immediate Generator مسئول استخراج و گسترش علامت‌دار (Sign-Extension) مقادیر فوری (Immediate) از دستورالعمل‌های RISC-V است. این مقدار فوری برای بسیاری از دستورالعمل‌های فرمت I و S کاربرد دارد و در اجرای درست عملیات‌هایی مانند addi، lw، sw و غیره حیاتی است.

ورودی‌ها:

Instr : داده ۳۲ بیتی دستور RISC-V (شامل کل فیلدهای دستور).

ImmSel : سیگنال یک‌بیتی برای تعیین نوع فوری («۰»: I-Type، «1»: S-Type).

خروجی:

Imm : مقدار فوری ۳۲ بیتی (با Sign-Extension صحیح از بخش مربوط دستور).

در حالت I-Type (ImmSel='0'):

مقدار فوری ۱۲ بیت از بیت‌های [۳۱:۲۰] استخراج می‌شود.

اگر بیت ۳۱ (بیت علامت) '۰' باشد، بالاترین ۲۰ بیت صفر می‌شوند (Zero Extension).

اگر بیت ۳۱ برابر '۱' باشد، بالاترین ۲۰ بیت با '۱' پر می‌شوند (Sign Extension).

در حالت S-Type (ImmSel='1'):

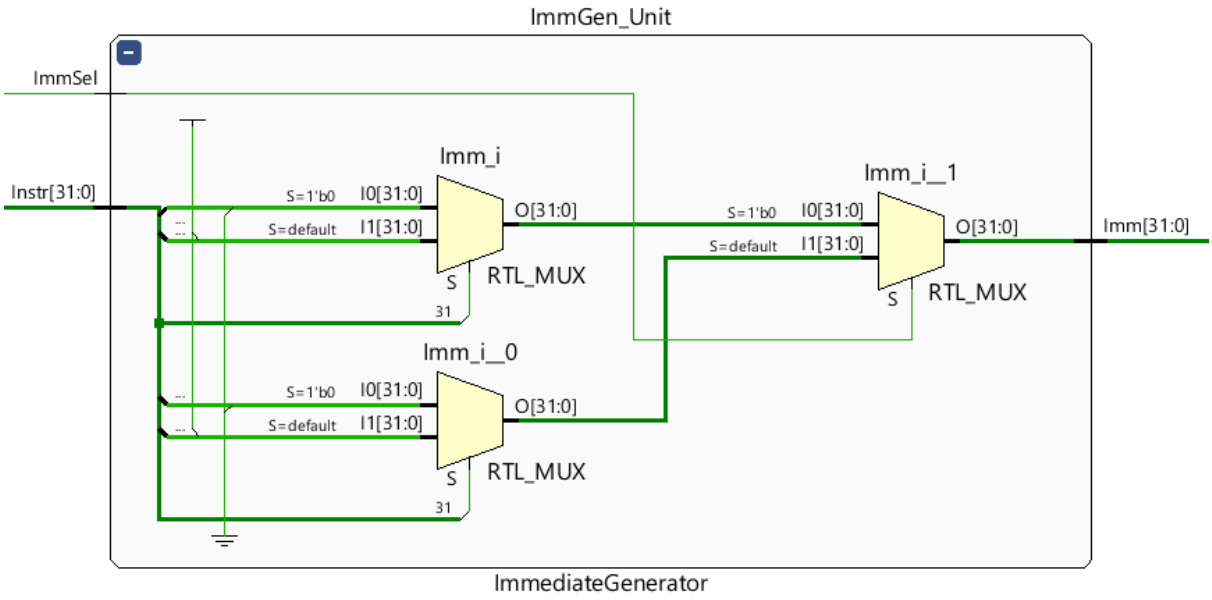
مقدار فوری از کنار هم گذاشتن بیت‌های [۳۱:۲۵] و [۱۱:۷] ساخته می‌شود (ترتیب: (Instr(31 downto 25 & (Instr(11 downto 7).

مشابه بالا، گسترش علامت مطابق بیت ۳۱ انجام می‌شود.

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/ImmediateGenerator.vhd

```
use IEEE.STD_LOGIC_1164.ALL;

-- Entity declaration for Immediate Generator
entity ImmediateGenerator is
    Port (
        Instr   : in  std_logic_vector(31 downto 0);   -- Input: 32-bit instruction word
        ImmSel  : in  std_logic;                       -- Immediate type selector ('0': I-type, '1': S-type)
        Imm     : out std_logic_vector(31 downto 0)    -- Output: 32-bit sign-extended immediate
    );
end ImmediateGenerator;

architecture Behavioral of ImmediateGenerator is
begin
    process(Instr, ImmSel)
    begin
        if ImmSel='0' then -- I-type immediate format: bits [31:20], sign-extended
            if Instr(31)='0' then
                Imm <= X"00000" & Instr(31 downto 20);  -- Positive: zero extend upper bits
            else
                Imm <= X"FFFFF" & Instr(31 downto 20);  -- Negative: one (sign) extend
            end if;
        else -- S-type immediate format: bits [31:25] & [11:7], sign-extended
            if Instr(31)='0' then
                Imm <= X"00000" & Instr(31 downto 25) & Instr(11 downto 7);
            else
                Imm <= X"FFFFF" & Instr(31 downto 25) & Instr(11 downto 7);
            end if;
        end if;
    end process;
end Behavioral;
```

استخراج صحیح فیلد فوری در فرمت‌های I/S، کاملاً مطابق استاندارد RISC-V انجام می‌شود. برای تطابق با فرمت ۳۲ بیتی، همیشه ۲۰ بیت بالایی (بسته به علامت) با صفر یا یک پر می‌شوند.ساختار شرطی ساده و خوانا، پیاده‌سازی را قابل گسترش برای فرمت‌های دیگر نیز کرده است. امکان دریافت همزمان هر نوع دستور I و S بدون نیاز به ماژول جداگانه.



واحد حساب و منطق (ALU)

واحد ALU (Arithmetic Logic Unit) بخش اصلی پردازنده برای انجام عملیات‌های حسابی (جمع، تفریق) و منطقی (AND، OR) است. این بخش با دریافت دو ورودی ۳۲ بیتی و یک سیگنال انتخاب، نتیجه عملیات را در خروجی ۳۲ بیتی تولید می‌کند.

ورودی‌ها:

A: اپراند اول، ۳۲ بیت

B: اپراند دوم، ۳۲ بیت

ALUSel: سیگنال انتخاب عملیات ۲ بیتی

"۰۰": جمع (ADD)

"۰۱": تفریق (SUB)

"۱۰": AND (عملیات منطقی)

"۱۱": OR (عملیات منطقی)

خروجی:

Result: خروجی ۳۲ بیتی نتیجه عملیات

با توجه به مقدار ALUSel، عملیات مورد نظر روی ورودی‌های A و B انجام می‌شود: جمع (ADD): تبدیل ورودی‌ها به نوع signed و جمع آن‌ها با یکدیگر , تفریق (SUB): تبدیل ورودی‌ها به نوع signed و تفریق

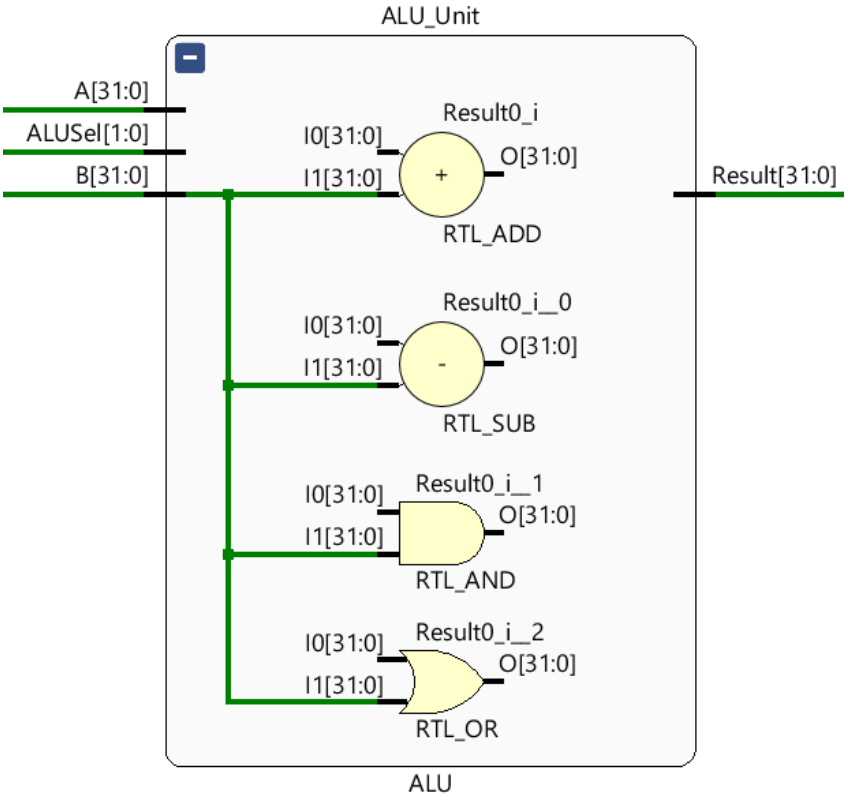
AND, (عملیات منطقی): هر بیت از دو ورودی با هم AND می‌شود

OR(عملیات منطقی): هر بیت از دو ورودی با هم OR می‌شود ,خروجی ماژول دقیقاً مطابق با نتیجه عملیات انتخابی (با پهنای باند کامل ۳۲ بیت) تولید می‌شود.

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/ALU.vhd

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

-- Entity declaration for Arithmetic Logic Unit (ALU)
entity ALU is
    Port (
        A      : in  std_logic_vector(31 downto 0);       -- First 32-bit operand
        B      : in  std_logic_vector(31 downto 0);       -- Second 32-bit operand
        ALUSel : in  std_logic_vector(1 downto 0);        -- ALU operation selector
        -- "00": ADD, "01": SUB, "10": AND, "11": OR
        Result : out std_logic_vector(31 downto 0)        -- 32-bit result output
    );
end ALU;

architecture Behavioral of ALU is
begin
    process(A, B, ALUSel)
    begin
        case ALUSel is
            when "00" =>
                Result <= std_logic_vector(signed(A) + signed(B));  -- ADD operation
            when "01" =>
                Result <= std_logic_vector(signed(A) - signed(B));  -- SUBTRACT operation
            when "10" =>
                Result <= A and B;                                  -- AND operation (bitwise)
            when others =>
                Result <= A or B;                                   -- OR operation (bitwise)
        end case;
    end process;
end Behavioral;
```

تبدیل سیگنال‌های ورودی از std_logic_vector به نوع signed برای جمع و تفریق مطابق استاندارد VHDL، مناسب برای پردازش اعداد منفی.عملیات منطقی معمولا بر روی بیت‌های std_logic_vector انجام می‌شود و نیازی به تبدیل نوع داده‌ای ندارد. انتخاب عملیات توسط یک سیگنال ۲ بیتی ساده انجام می‌شود که قابلیت توسعه عملیات‌های بیشتر نیز وجود دارد. در صورتی که مقدار ALUSel خارج از مقادیر تعریف‌شده باشد، عملیات OR اجرا می‌شود (پیش‌فرض when others).



حافظه داده (Data Memory)

ماژول **Data Memory** وظیفه نگهداری و دسترسی به داده‌های بارگذاری و ذخیره شده توسط پردازنده (Load/Store) را دارد. این واحد به صورت byte-addressable پیاده‌سازی شده و قابلیت خواندن/نوشتن داده ۳۲ بیتی (۴ بایت) را داراست.

ورودی‌ها:

Clk : سیگنال کلاک

MemRW : سیگنال کنترل خواندن/نوشتن حافظه (۱ برای write, ۰ برای read)

Addr : آدرس ۶ بیتی (برای انتخاب محل داده در حافظه: ۰ تا ۶۳، ولی برای word باید ۰ تا ۱۲۴ در نظر گرفت)

DataW : داده‌ی ۳۲ بیتی (ورودی برای عملیات ذخیره / Store)

خروجی‌ها:

DataR : داده‌ی ۳۲ بیتی خواند شده از حافظه (خروجی برای عملیات بارگذاری / Load)

حافظه:

از نوع آرایه ۱۲۸ بایتی (dmem_type)، هر خانه ۸ بیت (byte-addressable)

نوشتن داده:

در لبه بالارونده کلاک (if rising_edge(Clk))، اگر 'MemRW='1 و آدرس معتبر باشد، ۴ بایت متوالی از سیگنال ورودی DataW به ترتیب در خانه‌های Addr تا Addr+3 نوشته می‌شود.

توجه: داده به صورت **Big Endian** (بایت پرارزش‌تر در اندیس بیشتر) ذخیره می‌شود.

خواندن داده:

داده خوانده‌شده همیشه فعال است. حاصل ترکیب ۴ خانه متوالی حافظه (از آدرس Addr تا Addr+3) و ارسال آن به خروجی.

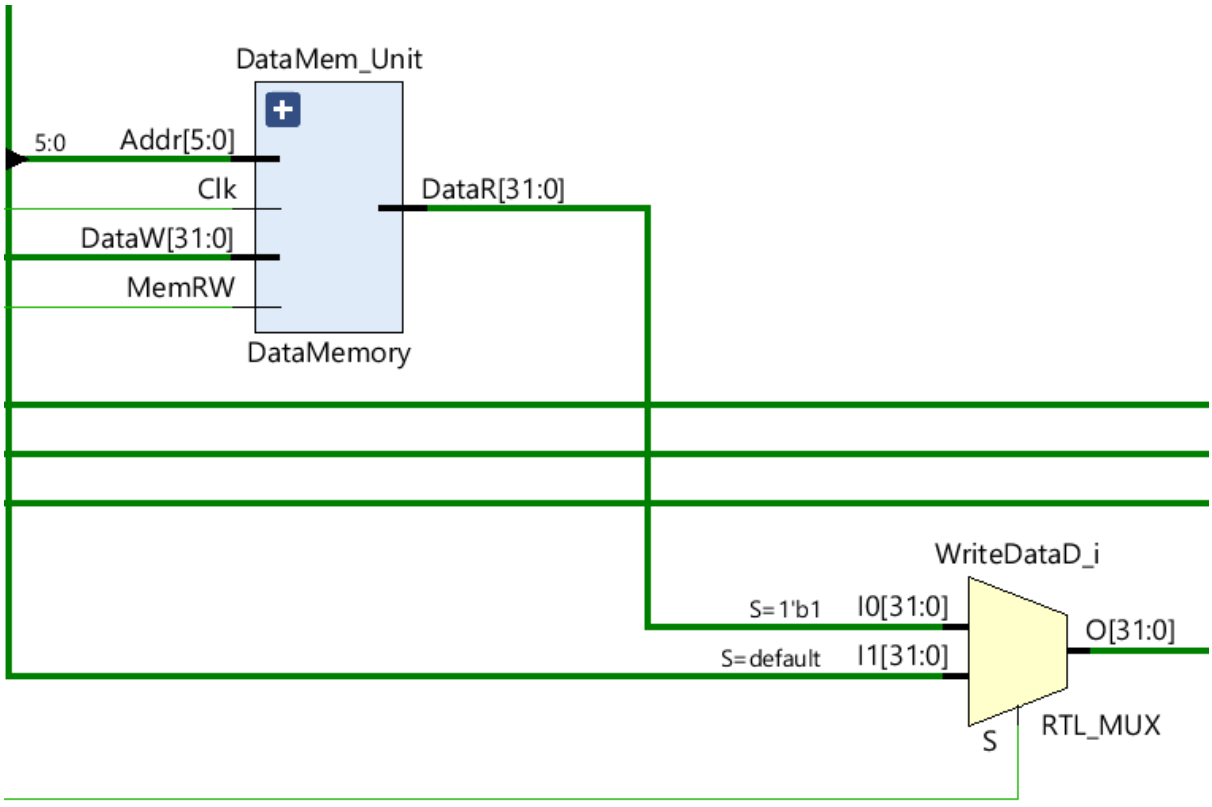
داده بر اساس ترتیب بایت‌ها به صورت [Byte3][Byte2][Byte1][Byte0] به خروجی متصل می‌شود.

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/DataMemory.vhd

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

-- Entity declaration for Data Memory
entity DataMemory is
    Port (
        Clk    : in  std_logic;                     -- Clock signal
        MemRW  : in  std_logic;                     -- Memory Read/Write control ('1': write, '0': read)
        Addr   : in  std_logic_vector(5 downto 0);  -- Memory address input (6 bits)
        DataW  : in  std_logic_vector(31 downto 0); -- Data input for store (SW)
        DataR  : out std_logic_vector(31 downto 0)  -- Data output for load (LW)
    );
end DataMemory;

architecture Behavioral of DataMemory is
    -- Define an array for 128 bytes of data memory (byte-addressable)
    type dmem_type is array (0 to 127) of std_logic_vector(7 downto 0);
    signal dmem : dmem_type := (others => (others => '0'));  -- Initialize all bytes to zero

    -- Internal integer address for indexing (byte level)
    signal addr_int : integer range 0 to 124;
begin
    addr_int <= to_integer(unsigned(Addr));  -- Convert input address to integer

    process(Clk)
    begin
        if rising_edge(Clk) then
            if MemRW='1' then  -- Memory Write (Store Word - SW)
                if addr_int <= 124 then
                    dmem(addr_int+0) <= DataW(7 downto 0);      -- Store byte 0 (LSB)
```

```
                    dmem(addr_int+1) <= DataW(15 downto 8);     -- Store byte 1
                    dmem(addr_int+2) <= DataW(23 downto 16);    -- Store byte 2
                    dmem(addr_int+3) <= DataW(31 downto 24);    -- Store byte 3 (MSB)
                end if;
            end if;
        end if;
    end process;

    -- Data read: Concatenate 4 consecutive bytes (Big-Endian order) into 32-bit word
    DataR <= dmem(addr_int + 3) & dmem(addr_int + 2) & dmem(addr_int + 1) & dmem(addr_int + 0);
end Behavioral;

```

حافظه داده به صورت ۱۲۸ بایت پیاده شده و به کمک تبدیل آدرس ۶ بیتی به عدد صحیح برای دسترسی داخلی استفاده می‌کند. امکان نوشتن/خواندن ۳۲ بیت با ترتیب صحیح بایت‌ها فراهم شده است. این ساختار همخوان با دستورات lw (load word) و sw (store word) در معماری RISC-V است. همیشه عملیات خواندن فعال است و نیازی به سیگنال خاص جهت خواندن نیست؛ تنها برای نوشتن، فعال‌سازی لازم است. در صورت ارجاع به آدرسی خارج از بازه معتبر، عملیات نوشتن صورت نمی‌گیرد.



فایل رجیستر (Register File)

ماژول **Register File** حافظه‌ای سریع و کوچک متشکل از ۳۲ رجیستر ۳۲ بیتی است که نقش حیاتی در ذخیره داده‌های موقت، مقادیر میانجی، و مسیرهای کنترلی پردازنده دارد. این واحد دسترسی همزمان به دو رجیستر برای خواندن و یک رجیستر برای نوشتن را فراهم می‌کند.

ورودی/خروجی‌ها:

ورودی‌ها:

Clk : سیگنال کلاک برای سنکرون کردن عملیات نوشتن

Reset : سیگنال ریست فعال بالا (پاک‌سازی تمامی رجیسترها)

AddrA : آدرس رجیستر خواندنی A (۵ بیت)

AddrB : آدرس رجیستر خواندنی B (۵ بیت)

AddrD : آدرس رجیستر نوشتنی D (۵ بیت)

DataD : داده‌ی ۳۲ بیتی برای نوشتن

RegWEn : سیگنال فعال‌ساز نوشتن (نوشتن تنها زمانی انجام می‌شود که این سیگنال فعال و آدرس هدف غیر x0 باشد)

خروجی‌ها:

DataA : داده خوانده‌شده از رجیستر A

DataB : داده خوانده‌شده از رجیستر B

x1_out تا x5_out : خروجی مستقیم رجیسترهای x1 تا x5 (قابل استفاده جهت مشاهده و دیباگ در شبیه‌سازی)

نوشتن: در هر لبه بالارونده کلاک، اگر Reset فعال باشد، تمامی رجیسترها صفر می‌شوند. در غیر این صورت، اگر سیگنال RegWEn فعال و آدرس مقصد غیر از x0 باشد، داده جدید در رجیستر مقصد نوشته می‌شود (مطابق قانون RISC-V که x0 همیشه صفر می‌ماند و غیرقابل تغییر است).

خواندن: داده‌های خوانده‌شده از آدرس‌های AddrA و AddrB، همیشه از همان لحظه بر روی خروجی‌ها (DataA و DataB) قابل مشاهده‌اند.

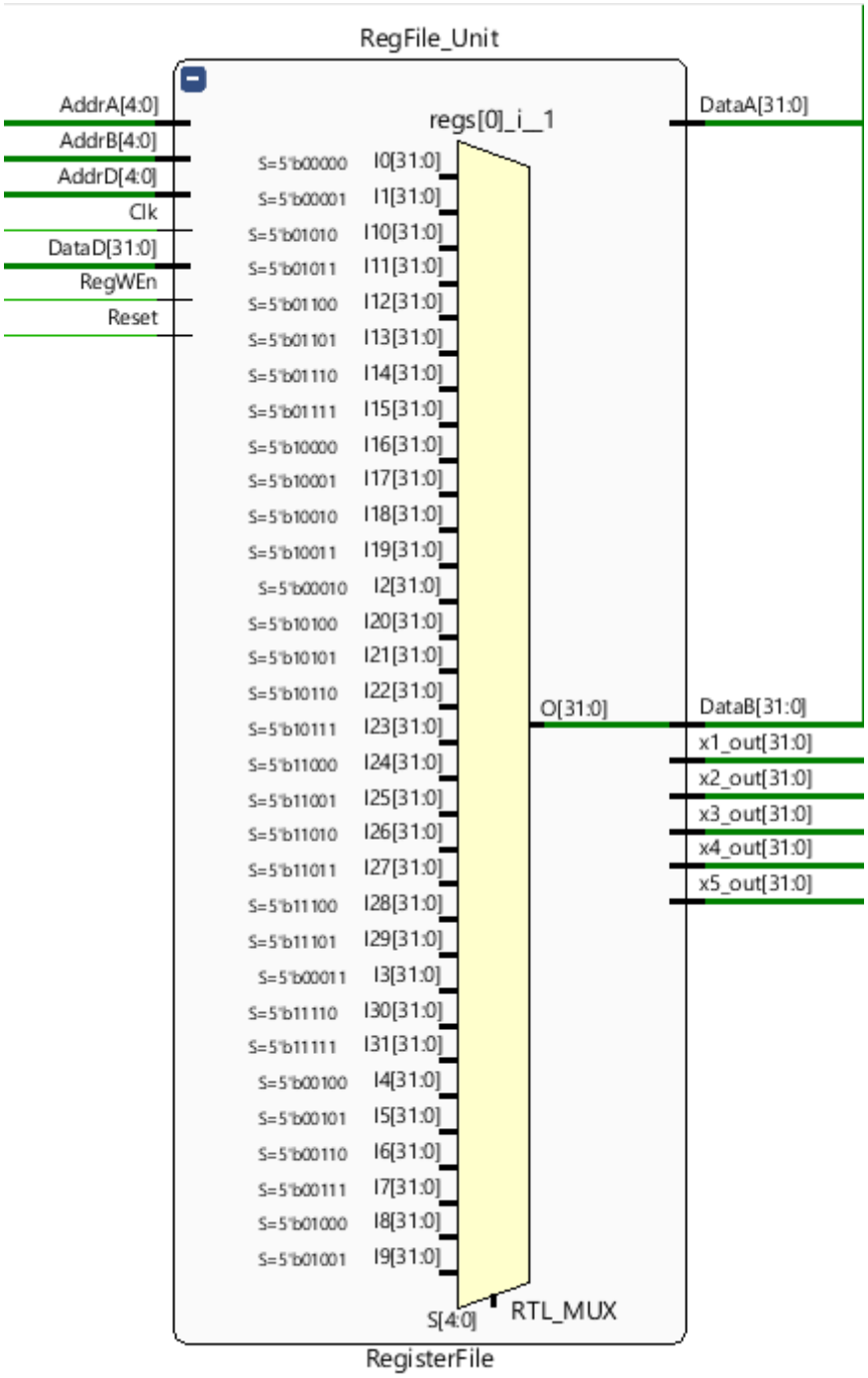
خروجی برای دیباگ: مقادیر فعلی رجیسترهای x1 تا x5 همواره به خروجی‌های اختصاصی هدایت می‌شوند برای بررسی وضعیت رجیسترها در شبیه‌سازی یا دیباگ.

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/RegisterFile.vhd

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

-- Entity declaration for Register File (32 x 32-bit registers)
entity RegisterFile is
    Port (
        Clk     : in  std_logic;                     -- Clock signal
        Reset   : in  std_logic;                     -- Reset (active high)
        AddrA   : in  std_logic_vector(4 downto 0);  -- Read address for output A
        AddrB   : in  std_logic_vector(4 downto 0);  -- Read address for output B
        AddrD   : in  std_logic_vector(4 downto 0);  -- Write address
        DataD   : in  std_logic_vector(31 downto 0); -- Data to write
        RegWEn  : in  std_logic;                     -- Register write enable
        DataA   : out std_logic_vector(31 downto 0); -- Read data from Reg[AddrA]
        DataB   : out std_logic_vector(31 downto 0); -- Read data from Reg[AddrB]
        x1_out  : out std_logic_vector(31 downto 0); -- Direct output: Reg[1]
        x2_out  : out std_logic_vector(31 downto 0); -- Direct output: Reg[2]
        x3_out  : out std_logic_vector(31 downto 0); -- Direct output: Reg[3]
        x4_out  : out std_logic_vector(31 downto 0); -- Direct output: Reg[4]
        x5_out  : out std_logic_vector(31 downto 0)  -- Direct output: Reg[5]
    );
end RegisterFile;

architecture Behavioral of RegisterFile is
    -- Array of 32 registers, 32 bits each, initialized to zero
    type reg_array is array (0 to 31) of std_logic_vector(31 downto 0);
    signal regs : reg_array := (others => (others => '0'));
begin
    process(Clk)
    begin
        if rising_edge(Clk) then
            if Reset = '1' then
                regs <= (others => (others => '0'));          -- Clear all registers on reset
            elsif RegWEn = '1' and AddrD /= "00000" then
                regs(to_integer(unsigned(AddrD))) <= DataD;  -- Write to register AddrD if enabled (except x0)
            end if;
        end if;
    end process;

    -- Read data from register AddrA and AddrB
    DataA <= regs(to_integer(unsigned(AddrA)));   -- Read port A
    DataB <= regs(to_integer(unsigned(AddrB)));   -- Read port B

    -- Direct outputs for registers x1 to x5 (for debug/monitoring)
    x1_out <= regs(1);
    x2_out <= regs(2);
    x3_out <= regs(3);
    x4_out <= regs(4);
    x5_out <= regs(5);
end Behavioral;
```

رجیستر شماره صفر (x0) طبق استاندارد RISC-V غیرقابل‌نوشتن و همواره مقدار صفر دارد. پیاده‌سازی رجیسترها به صورت آرایه‌ای با مقدار اولیه صفر. قابلیت ریست‌پذیری کامل. هر دو پورت خواندنی کاملاً

مستقل و موازی هستند. پنج خروجی مجزا برای رجیسترهای ۱ تا ۵ به منظور تحلیل نتایج برنامه و تست عملکرد در محیط شبیه‌سازی.



واحد کنترل (Control Unit)

واحد کنترل (Control Unit) یکی از اجزای کلیدی پردازنده تک‌چرخه است و وظیفه آن **رمزگشایی (Decode)** دستورالعمل دریافتی و تولید سیگنال‌های کنترلی مناسب جهت هدایت جریان داده و عملیات اجرایی در سایر اجزای پردازنده است. این واحد با بررسی باینری دستور و

فیلدهای اپکد (opcode)، فانکشن ۳ (funct3) و فانکشن ۷ (funct7)، نوع عملیات (حسابی، منطقی، load, store) را تشخیص داده و سیگنال‌های لازم را ست می‌کند.

ورودی و خروجی‌ها:

ورودی:

Instr : دستورالعمل ۳۲ بیتی کد ماشین RISC-V

خروجی:

ALUSel : انتخاب نوع عملیات در ALU (۲ بیت: ۰۰=جمع، ۰۱=تفریق، ۱۰=AND، ۱۱=OR)

BSel : انتخاب ورودی دوم ALU (۰=مقدار رجیستر، ۱=مقدار Immediate)

RegWEn : فعال‌ساز نوشتن در فایل رجیستر (۱=فعال)

MemRW : فعال‌ساز نوشتن حافظه داده (۱=ذخیره‌سازی / SW)

WBSel : انتخاب مقدار برگشتی به فایل رجیستر (۰=خروجی ALU، ۱=خروجی Data Memory)

ImmSel : انتخاب نوع Immediate (۰=فرمت I، ۱=فرمت S)

با استخراج فیلدهای کلیدی (opcode، funct3 و funct7) از دستور، نوع کلی و جزئی عملکرد دستورالعمل تعیین می‌شود. سیگنال‌های خروجی پیش‌فرض به حالت خنثی (NOP) مقداردهی اولیه می‌شوند و سپس مطابق نوع دستور به صورت هدفمند مقداردهی می‌شوند:

دستورهای R-type (مانند add, sub, and, or):

رجیستر مقصد فعال، ورودی دوم ALU از رجیستر، و عملگر ALU بر اساس funct3 و funct7 تنظیم می‌شود.

دستورهای I-type (مانند addi, andi, ori):

رجیستر مقصد فعال، ورودی دوم ALU از immediate، نوع عملیات ALU و نوع Immediate انتخاب می‌شود.

بارگذاری کلمه (LW):

رجیستر مقصد فعال، آدرس از جمع رجیستر و Immediate، مقدار برگشتی از حجم حافظه داده انتخاب می‌شود.

ذخیره‌سازی کلمه (SW):

فقط نوشتن حافظه داده فعال، سایر سیگنال‌ها مقدار مناسب می‌گیرند.

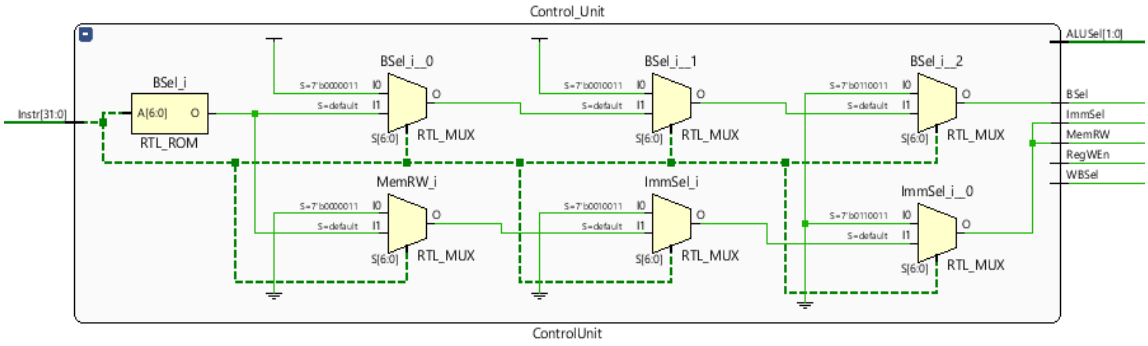
هر دستور تنها به یک مسیر و ماژول اصلی اشاره دارد و سیگنال‌های غیر نیازمند در همان سیکل غیرفعال هستند.

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/ControlUnit.vhd

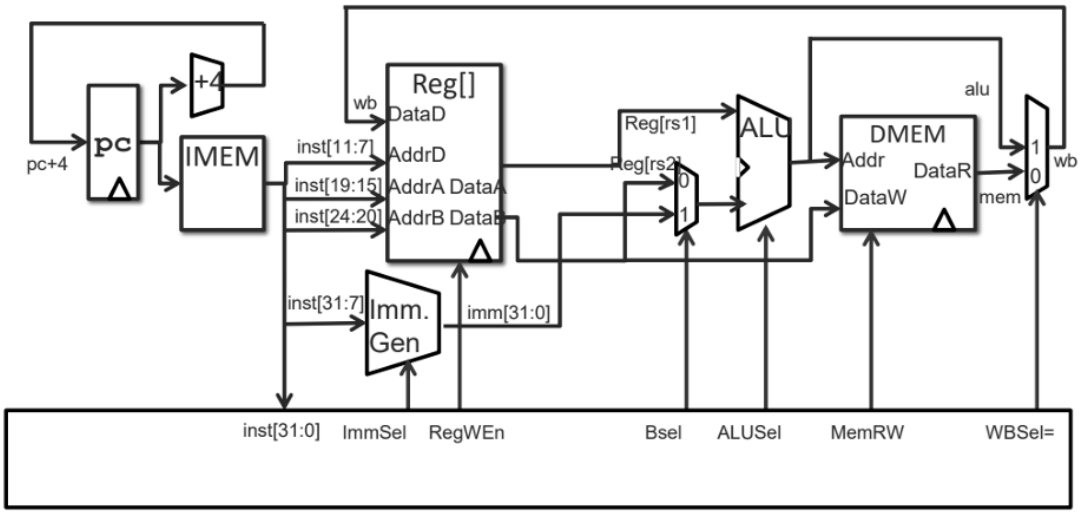
```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  -- Entity declaration: Control Unit for instruction decoding
5  entity ControlUnit is
6      Port (
7          Instr    : in  std_logic_vector(31 downto 0); -- Input: 32-bit instruction
8          ALUSel   : out std_logic_vector(1 downto 0);  -- ALU operation select
9          BSel     : out std_logic;                     -- ALU B input select (0: RegB, 1: immediate)
10         RegWEn   : out std_logic;                     -- Register write enable (1: write)
11         MemRW    : out std_logic;                     -- Data memory write enable (1: SW)
12         WBSel    : out std_logic;                     -- Write-back select (1: DataMem, 0: ALU)
13         ImmSel   : out std_logic                      -- Immediate type select (0: I-type, 1: S-type)
14     );
15 end ControlUnit;
16
17 architecture Behavioral of ControlUnit is
18     -- Internal signals for instruction decoding fields
19     signal opcode : std_logic_vector(6 downto 0);   -- Instruction[6:0]: opcode field
20     signal funct3 : std_logic_vector(2 downto 0);   -- Instruction[14:12]: funct3 field
21     signal funct7 : std_logic_vector(6 downto 0);   -- Instruction[31:25]: funct7 field
22 begin
23     -- Assign opcode, funct3, and funct7 for easier decoding
24     opcode <= Instr(6 downto 0);
25     funct3 <= Instr(14 downto 12);
26     funct7 <= Instr(31 downto 25);
27
28     process(opcode, funct3, funct7)
29     begin
30         -- Default values for control signals (NOP)
31         ALUSel <= "00";
```

```
        -- Default values for control signals (NOP)
        ALUSel <= "00";
        BSel   <= '0';
        RegWEn <= '0';
        MemRW  <= '0';
        WBSel  <= '0';
        ImmSel <= '0';
        -- R-type instruction (add, sub, and, or)
        if (opcode = "0110011") then
            RegWEn <= '1';      -- Enable register write
            BSel   <= '0';      -- Use RegB for ALU input B
            case funct3 is
                when "000" =>   -- ADD/SUB
                    if funct7="0000000" then
                        ALUSel <= "00"; -- ADD
                    else
                        ALUSel <= "01"; -- SUB
                    end if;
                when "111" =>   -- AND
                    ALUSel <= "10";
                when "110" =>   -- OR
                    ALUSel <= "11";
                when others => null;
            end case;
        -- I-type arithmetic (addi, andi, ori)
        elsif (opcode = "0010011") then
            RegWEn <= '1';      -- Enable register write
            BSel   <= '1';      -- Use immediate for ALU input B
            case funct3 is
                when "000" =>   -- ADDI
                    ALUSel <= "00";
                when "111" =>   -- ANDI
                    ALUSel <= "10";
                when "110" =>   -- ORI
                    ALUSel <= "11";
                when others => null;
            end case;
            ImmSel <= '0';      -- Select I-type immediate
        -- Load word (LW)
        elsif (opcode = "0000011") then
            RegWEn <= '1';      -- Enable register write
            BSel   <= '1';      -- Use immediate for ALU input B (address offset)
            ALUSel <= "00";     -- ALU add for address calculation
            MemRW  <= '0';      -- No memory write
            WBSel  <= '1';      -- Write-back from Data Memory
            ImmSel <= '0';      -- I-type immediate format
        -- Store word (SW)
        elsif (opcode = "0100011") then
            RegWEn <= '0';      -- No register write
            BSel   <= '1';      -- Use immediate for ALU input B (address offset)
            ALUSel <= "00";     -- ALU add for address calculation
            MemRW  <= '1';      -- Memory write
            WBSel  <= '0';      -- (Unused for SW)
            ImmSel <= '1';      -- S-type immediate format
        end if;
    end process;
end Behavioral;
```

شناسایی دستور با مقایسه دقیق فیلد فوقانی (opcode) و براساس نوع دستور، سیگنال‌های جزئی (funct3/7) تحلیل می‌شوند. خروجی ALUSel دو بیتی، کانال عملیات ALU را با توجه به نوع هر عملکرد انتخاب می‌کند. تمامی سیگنال‌های جانبی مانند WBSel و ImmSel نیز با دقت و بر اساس ماهیت دستور ست می‌شوند. طراحی کد انعطاف‌پذیر است و امکان افزودن دستور یا عملکرد (همچون branchها) به آسانی قابل گسترش می‌باشد.



اتصال اجزای معماری (System Integration / Data Path)



شرح نحوه ارتباط ماژول‌ها در TopCPU

در طراحی پردازنده RISC-V تک‌چرخه، ماژول‌های اصلی (Program Counter، Instruction Memory، Register File، ALU، Data Memory، Immediate Generator و Control Unit) به صورت ساختاری در ماژول بالادستی **TopCPU** به یکدیگر متصل می‌شوند. هر چرخه کلاک، یک دستور به طور کامل از مرحله واکشی تا اجرا و بازگشت نتیجه به رجیسترها طی می‌کند. نحوه اتصال ماژول‌ها و تبادل سیگنال‌ها به شرح زیر است:

C:/Users/hosse/Dropbox/PC/Desktop/RISC_V/section_3/3/project_6/project_6.srcs/sources_1/new/TopCPU.vhd

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

-- Entity declaration for the top-level RISC-V CPU
entity TopCPU is
    Port (
        Clk    : in  std_logic;                          -- System clock input
        Reset  : in  std_logic;                          -- Asynchronous reset input
        x1_out : out std_logic_vector(31 downto 0);      -- Debug output: register x1
        x2_out : out std_logic_vector(31 downto 0);      -- Debug output: register x2
        x3_out : out std_logic_vector(31 downto 0);      -- Debug output: register x3
        x4_out : out std_logic_vector(31 downto 0);      -- Debug output: register x4
        x5_out : out std_logic_vector(31 downto 0)       -- Debug output: register x5
    );
end TopCPU;

architecture Structural of TopCPU is
    -- Internal signals for datapath and control
    signal PC_in, PC_out            : std_logic_vector(5 downto 0);      -- Program Counter input/output
    signal Instr                    : std_logic_vector(31 downto 0);     -- Current fetched instruction
    signal Imm                      : std_logic_vector(31 downto 0);     -- Immediate value from the generator
    signal RegA, RegB               : std_logic_vector(31 downto 0);     -- Register file read ports
    signal AddrA, AddrB, AddrD      : std_logic_vector(4 downto 0);      -- Register addresses (rs1, rs2, rd)
    signal ALU_A, ALU_B, ALU_Result : std_logic_vector(31 downto 0);     -- ALU operands, result
    signal ALUSel                   : std_logic_vector(1 downto 0);      -- ALU operation selection
    signal BSel, RegWEn             : std_logic;                         -- Control: B mux, reg write enable
    signal MemRW, WBSel, ImmSel     : std_logic;                         -- Control: mem write, wb select, imm type
    signal DataMemR                 : std_logic_vector(31 downto 0);     -- Data read from memory
    signal WriteDataD               : std_logic_vector(31 downto 0);     -- Data to be written into the register fi
begin
    -- Extract register addresses from instruction (RISC-V standard encoding)
    AddrA <= Instr(19 downto 15);              -- rs1: source register 1
    AddrB <= Instr(24 downto 20);              -- rs2: source register 2
    AddrD <= Instr(11 downto 7);               -- rd: destination register

    -- Program Counter unit: Generates PC_out, updates by PC_in (PC+4 per cycle)
    PC_Unit: entity work.ProgramCounter
        port map (
            Clk    => Clk,
            Reset  => Reset,
            PC_in  => PC_in,                   -- Next program counter value
            PC_out => PC_out                   -- Current program counter value
        );

    PC_in <= std_logic_vector(unsigned(PC_out) + 4);    -- Increment PC by 4 every cycle

    -- Instruction Memory unit: Fetches instruction at PC address
    IMEM_Unit: entity work.InstructionMemory
        port map (
            Addr    => PC_out,                 -- PC address to fetch from
            InstOut => Instr                   -- Output: fetched instruction
        );

    -- Immediate Generator unit: Extracts/extends immediate based on instruction and control
    ImmGen_Unit : entity work.ImmediateGenerator
        port map (
            Instr  => Instr,                   -- Input: current instruction
            ImmSel => ImmSel,                  -- Immediate type selector (I/S)
            Imm    => Imm                      -- Output: sign-extended immediate
        );
```

```
66
67      -- ALU unit: Executes operation based on control unit (ALUSel)
68      ALU_Unit: entity work.ALU
69          port map (
70              A       => ALU_A,                   -- ALU operand A
71              B       => ALU_B,                   -- ALU operand B
72              ALUSel  => ALUSel,                  -- ALU operation select
73              Result  => ALU_Result               -- Output: ALU result
74          );
75
76      -- Data Memory unit: For load/store instructions (address = ALU result [5:0])
77      DataMem_Unit: entity work.DataMemory
78          port map (
79              Clk    => Clk,
80              MemRW  => MemRW,                    -- Memory write enable (SW)
81              Addr   => ALU_Result(5 downto 0),   -- Address for read/write (lower 6 bits)
82              DataW  => RegB,                     -- Data to store (SW)
83              DataR  => DataMemR                  -- Data loaded (LW)
84          );
85
86      -- Write back data selection: From DataMemory (LW) or ALU (normal operations)
87      WriteDataD <= DataMemR when WBSel='1' else ALU_Result;
88
89      -- Register File: Handles all register reads/writes
90      RegFile_Unit: entity work.RegisterFile
91          port map (
92              Clk     => Clk,
93              Reset   => Reset,
94              AddrA   => AddrA,                   -- rs1
95              AddrB   => AddrB,                   -- rs2
96              AddrD   => AddrD,                   -- rd
97              DataD   => WriteDataD               -- Data to write to rd
```

```
90      RegFile_Unit: entity work.RegisterFile
91          port map (
92              Clk     => Clk,
93              Reset   => Reset,
94              AddrA   => AddrA,                   -- rs1
95              AddrB   => AddrB,                   -- rs2
96              AddrD   => AddrD,                   -- rd
97              DataD   => WriteDataD,              -- Data to write to rd
98              RegWEn  => RegWEn,                  -- Reg write enable
99              DataA   => RegA,                    -- Read data (rs1)
100             DataB   => RegB,                    -- Read data (rs2)
101             x1_out  => x1_out,                  -- Debug: x1
102             x2_out  => x2_out,                  -- Debug: x2
103             x3_out  => x3_out,                  -- Debug: x3
104             x4_out  => x4_out,                  -- Debug: x4
105             x5_out  => x5_out                   -- Debug: x5
106         );
107
108     -- Control Unit: Decodes instruction, generates control signals for datapath
109     Control_Unit : entity work.ControlUnit
110         port map (
111             Instr   => Instr,                   -- Input: fetched instruction
112             ALUSel  => ALUSel,                  -- Output: ALU operation select
113             BSel    => BSel,                    -- ALU B mux select
114             RegWEn  => RegWEn,                  -- Reg file write enable
115             MemRW   => MemRW,                   -- Memory write enable
116             WBSel   => WBSel,                   -- Write-back data select
117             ImmSel  => ImmSel                   -- Immediate type/I or S
118         );
119 end Structural;
120
```

جریان اصلی داده و سیگنال‌ها

Program Counter (PC):

سیگنال **PC_out** آدرس فعلی را نگه می‌دارد و پس از هر سیکل با سیگنال **PC_in** (**PC+4**) یک دستور جدید را آدرس‌دهی می‌کند.

Instruction Memory:

با دریافت آدرس دستور فعلی از **PC_out**، سیگنال **Instr** (دستور ۳۲ بیتی) را تولید و برای رمزگشایی به سایر بخش‌ها می‌فرستد.

Control Unit:

با دریافت **Instr** سیگنال‌های کنترلی مانند **ALUSel**، **BSel**، **RegWEn**، **MemRW**، **WBSel**، **ImmSel** را برای تنظیم کل مسیر داده تولید می‌کند.

Register File:

آدرس‌های **AddrA** و **AddrB** (میدان rs1 و rs2) و **AddrD** (rd) را از دستور دریاف می‌کند و داده‌های **RegA** و **RegB** (خروجی دو پورت خواندن) را روی باس داخلی قرار می‌دهد.

نوشتن در رجیستر بر اساس **RegWEn** و تنها در صورت فعال بودن و غیربودن x0 انجام می‌شود.

Immediate Generator:

با توجه به نوع دستور (توسط سیگنال **ImmSel**)، مقدار immediate از فیلدهای مناسب استخراج و به صورت سیگنال **Imm** برای استفاده در ALU یا محاسبه آدرس ارسال می‌شود.

ALU:

داده ورودی A همیشه از **RegA** و ورودی B از بین **RegB** یا **Imm** (با توجه به سیگنال **BSel**) انتخاب می‌شود.

نوع عملیات (جمع، تفریق، AND، OR) با سیگنال **ALUSel** تعیین می‌گردد و خروجی **ALU_Result** برای استفاده مستقیم یا به عنوان آدرس داده (در LW/SW) به کار می‌رود.

Data Memory:

آدرس دسترسی به حافظه از **۶ بیت پایین ALU_Result** گرفته می‌شود.

برای SW مقدار **RegB** در آدرس مورد نظر قرار می‌گیرد (اگر **MemRW**=۱) و برای LW داده از حافظه به خروجی **DataMemR** فرستاده می‌شود.

Write Back:

مقدار برگشتی برای نوشتن در رجیستر یا خروجی ALU است (**WBSel=0**) یا خروجی حافظه داده (**WBSel=1**)، سیگنال **WriteDataD** تنظیم این مسیر را انجام می‌دهد و در سیکل بعد وارد فایل رجیستر می‌شود.

مزایا و نکات کلیدی ساختار تک‌چرخه

سادگی پیاده‌سازی:تمامی اجزا یک مسیر داده خطی و شفاف دارند و کنترل هر عملیات تنها با یک سیکل کلاک انجام می‌شود (سریع و طراحی آسان‌تر).

تاخیر کم (یک سیکل):هر دستور تنها طی یک سیکل از واکشی تا نوشتن رجیستر اجرا می‌شود (مناسب برای آموزش، قابلیت مشاهده آسان نتیجه دستورها).

عیب‌یابی و دیباگ راحت:دسترسی مستقیم به داده‌ی رجیسترها (x1...x5_out) برای تست و تحلیل عملکرد در شبیه‌سازی.

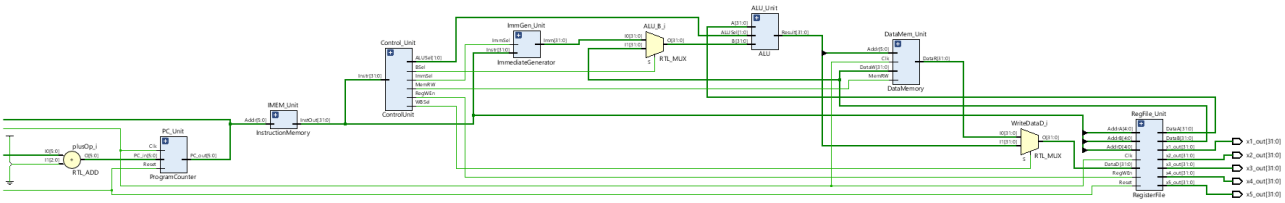
عدم وجود خطرهای ساختاری/وابستگی (structural/data hazards):چون همه چیز همزمان انجام می‌شود، نیازی به مکانیزم‌های پیچیده مثل فورواردینگ یا stall وجود ندارد.

قابلیت افزودن دستور جدیدمعماری انعطاف‌پذیر و قابل توسعه با حداقل تغییر در مسیر کنترل و داده.

معایب:با افزایش پیچیدگی و دستورات بیشتر، این ساختار دیگر در عمل راندمان کافی برای پروژه‌های صنعتی ندارد و قابلیت بهینه‌سازی ندارد.

دیاگرام مسیر داده (Data Path Diagram)

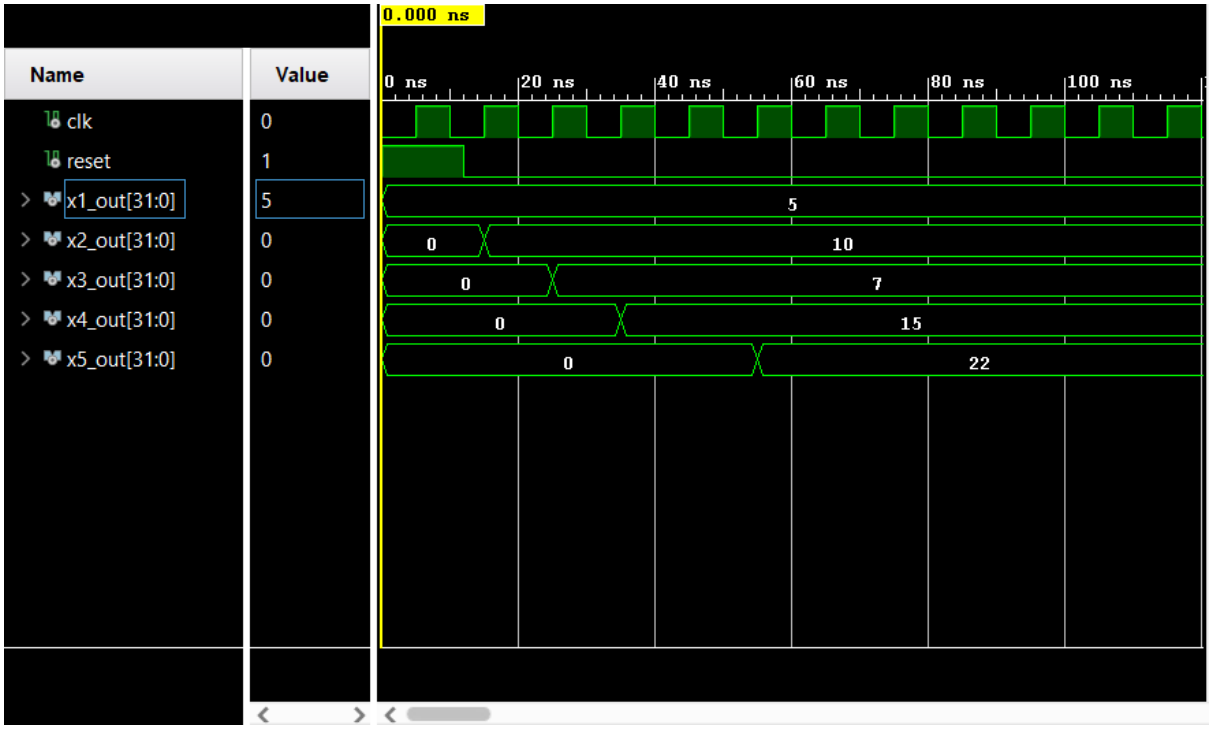
این دیاگرام ارتباط ماژول‌ها و مسیر عبور سیگنال‌ها را به صورت گرافیکی نمایش می‌دهد.



تست و شبیه‌سازی (Simulation & Test)

کد شماره ۱:

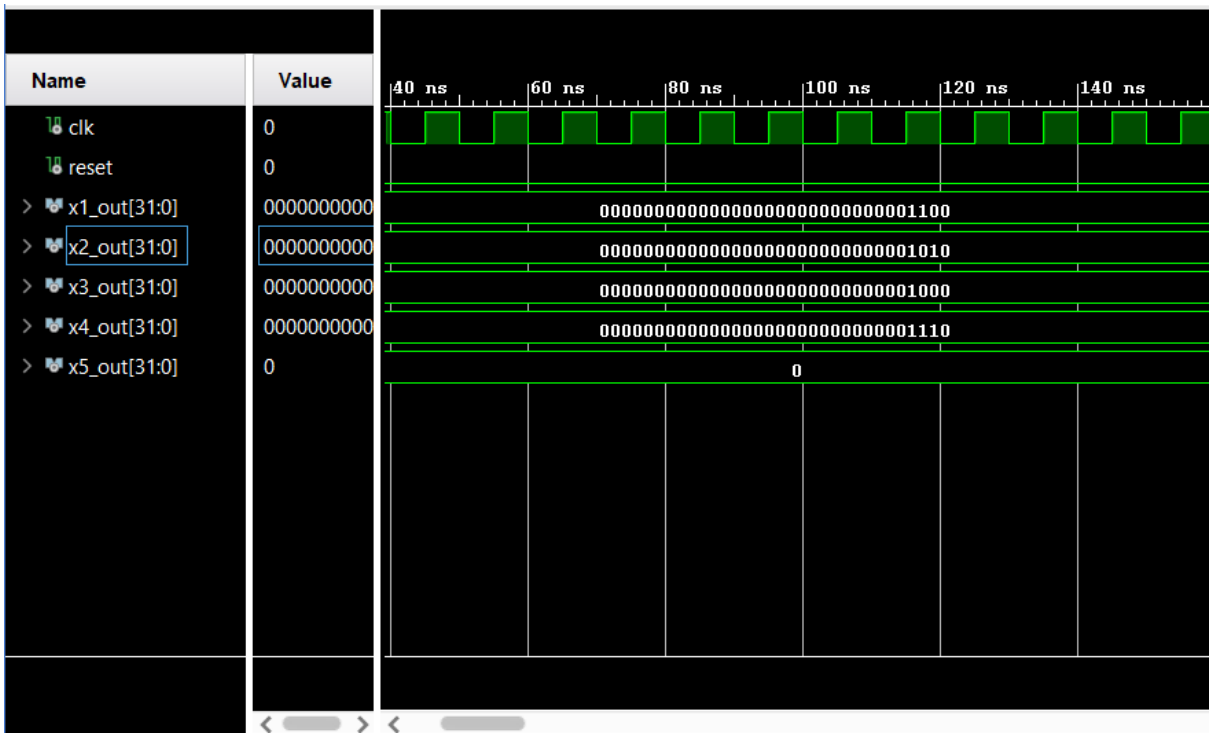
```
addi x1, x0, 5
addi x2, x0, 10
addi x3, x0, 7
add  x4, x1, x2
add  x5, x4, x3
```



در اولین کد تست، با استفاده از سه دستورaddi ، رجیسترهای x1 ، x2و x3 به‌ترتیب با مقادیر ۵، ۱۰ و ۷ مقداردهی شدند. در موج خروجی به‌وضوح دیده شد که با اجرای هر دستور مقدار مربوطه بدون تاخیر در رجیستر مقصد ثبت می‌شود. پس از آن، با اجرای دستورهایadd ، ابتدا مجموع x1 و x2 در x4 قرار گرفت، و سپس با جمع مقدار x4 وx3 ، مقدار نهایی ۲۲ در x5 ذخیره شد. موج شبیه‌سازی به خوبی این تغییر مقادیر را در بازه هر پالس کلاک نمایش داد؛ هر دستور که اجرا شد، بلافاصله مقدار رجیستر مربوطه عوض شد و هیچگونه تداخل یا هیزارد وجود نداشت. این موضوع نشانه صحت عملکرد مسیر داده و سامانه کنترل است.

کد شماره ۲:

```
addi x1, x0, 12
addi x2, x0, 10
and  x3, x1, x2
or   x4, x1, x2
```

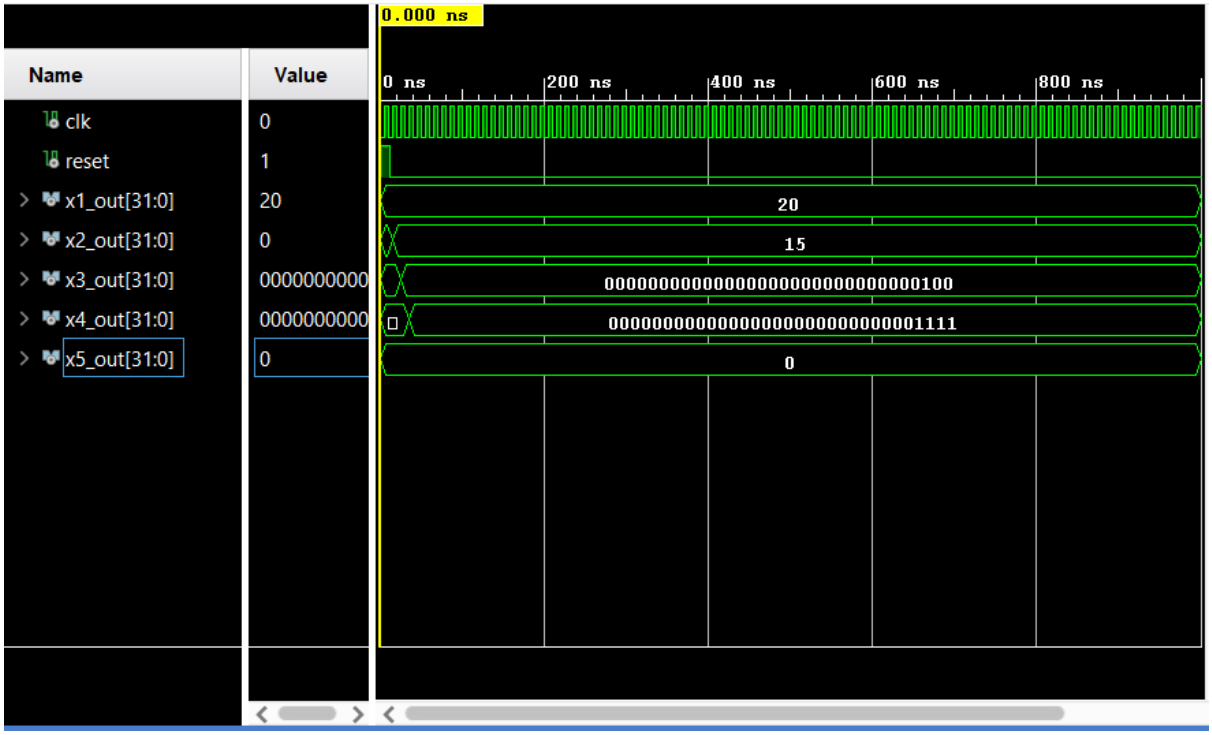


در دومین برنامه تست، عملکرد عملگرهای منطقی ALU با استفاده از دستور and و or روی مقادیر ثبت‌شده در رجیسترهای x1 و x2 بررسی شد. پس از مقداردهی x1 و x2 باaddi ، موج خروجی نشان داد مقدار ۱۲ در x1 و ۱۰ در x2 قرار گرفت. سپس با اجرای دستورand ، مقدار ۸ (AND بیتی ۱۲ و ۱۰) بدون تاخیر در x3 ظاهر شد و به‌دنبال آن، دستور or مقدار ۱۴ را در x4 ثبت کرد. با مرور دقیق موج

شبیه‌سازی، می‌توان مشاهده کرد که مقادیر خروجی رجیسترها دقیقاً طبق انتظار و کاملاً همزمان با هر دستور تغییر می‌کنند. بنابراین، پیاده‌سازی عملگرهای منطقی و کنترل سیگنال‌های ALUSel و MUX انتخاب‌گرها کاملاً درست بوده است.

کد شماره ۳:

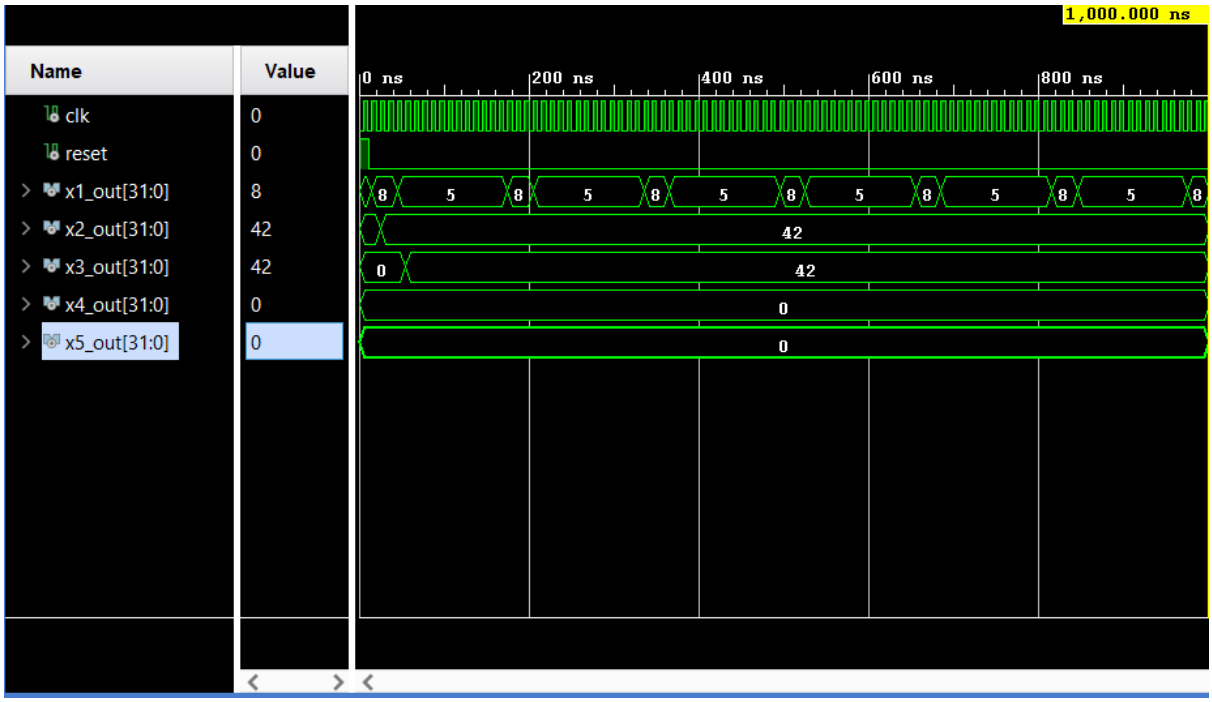
```
addi x1, x0, 20
addi x2, x1, -5
andi x3, x1, 0x0F
ori  x4, x2, 0x01
```



در کد تست سوم برای اطمینان از صحت پردازش Immediate ها و کنترل سیگنال‌های مرتبط با آنها، مجموعه‌ای از دستورهای addi ، andi و ori استفاده شده است. بر اساس موج خروجی، ابتدا مقدار ۲۰ به x1 وارد شده است. سپس، با اجرای addi x2, x1, -5 ، موج نشان می‌دهد که مقدار x2 به ۱۵ رسیده است. در ادامه و با اجرای andi x3, x1, 0x0F ، رجیستر x3 مقدار ۴) حاصل AND بیتی ۲۰ و ۱۵ (را گرفته و بلافاصله پس از اجرای دستور ori ، مقدار x4 به ۱۵ رسیده است. موج خروجی در این بخش به‌روشنی هر یک از این مراحل را نمایش می‌دهد و نشان می‌دهد منطق Immediate Generator و کنترل ALUSel/BSel در هر مرحله کاملاً دقیق و منطبق با انتظار کار کرده‌اند.

کد شماره ۴:(بخش امتیازی)

```
addi x1, x0, 8
addi x2, x0, 42
sw   x2, 0(x1)
addi x1, x1, -3
lw   x3, 3(x1)
```



در کد تست چهارم ، برای ارزیابی عملکرد مسیر Load/Store و حافظه داده، یک تست حافظه اجرا شد. موج خروجی شبیه‌سازی نشان داد که پس از مقداردهی x1 برابر با ۸ و x2 برابر با ۴۲، دستور sw به درستی مقدار x2 را در آدرس ۸ حافظه ذخیره کرد. سپس با اجرای addi x1, x1, -3 مقدار x1 به ۵ کاهش یافت و نهایتاً در موج دیده شد که دستور lw از آدرس ۳+۵=۸ مقدار ۴۲ را از حافظه خوانده و آن را در x3 قرار داده است. تغییر مقدار x3 روی موج کاملاً با لحظه اجرای دستور load منطبق است و صحت کامل پیاده‌سازی مسیر ارتباطی ALU ، حافظه داده و رجیستر فایل را نشان می‌دهد .در مجموع، تحلیل دقیق موج‌های شبیه‌سازی برای هر چهار تست بیانگر آن است که کلیه واحدهای پردازنده، سیگنال‌های کنترلی، گذرگاه‌های داده، و همچنین منطق حافظه و عملیات Immediate ، همگی بدون ایراد و با دقت کامل پیاده‌سازی شده‌اند. رفتار تک‌چرخه‌ای، به‌روزرسانی بلافاصله رجیسترها و نبود هیچگونه هیزارد یا

تداخل در کل اجرا به تایید می‌رسد و این موج‌های خروجی، صحت کامل پردازنده طراحی شده را تضمین می‌کند.