-

بسم الله الرحمن الرحیم

پروژه میانترم درس FPGA: پیاده‌سازی پردازنده Multi Cycle با معماری MIPS

اعضای گروه: زهرا لطیفی 9923069 – مریم مقتدری 9923073

توضیحات مربوط به انجام پروژه:

هدف از انجام این آزمایش آشنایی با مسیر داده و کنترلر در پردازنده ی MIPS و پیاده سازی توصیفی آن به صورت multi cycle- در زبان توصیف سخت افزار VHDL و تست آن توسط یک برنامه ی ساده با زبان اسمبلی MIPS است.

برای انجام آزمایش لازم است مطالب پیوست در رابطه با پردازنده MIPS از دانشگاه صنعتی شریف آقای دکتر موحدین در مورد مسیر داده و کنترلر برای ساختار cycle-multi به دقت مطالعه شوند.

آشنایی با پردازنده چندسیکلی در ساختار mips:

اکثر پردازنده ها معمولاً به دو بخش تقسیم می شوند: مسیر داده و واحد کنترل. مسیر داده مسیری است که پردازنده طی می کند تا یک دستورالعمل به طور کامل اجرا شود؛ این مسیر شامل تمام سخت افزار لازم برای انجام تمام عملیات های لازم است، که رجیسترها، مالتی‌پلکسرها، حافظه و ALU را شامل می‌شود.

همچنین هر دستورالعمل برای پردازش، مراحل مختلفی را طی خواهد کرد که اولین مرحله Fetching نام دارد. در این مرحله، یک دستورالعمل از حافظه وارد چرخه پردازش شده و PC برای اشاره به دستور بعدی به روز می شود. همچنین مقدار PC در یک رجیستر به نام "PC register" ذخیره می شود.

در مرحله بعد، دستورالعمل‌ها رمزگشایی می‌شوند (Decoding) و شش بیت ابتدایی از ورودی 32 بیتی، به عنوان opcode ذخیره می‌شود. در ادامه، این داده تعیین می‌کند که دستورالعمل از چه خانواده‌ای است (R-Type, I-Type, J-Type) و داده‌های دیگر مانند مقدار imm، رجیستر مبدا و مقصد و function را استخراج می‌کند.

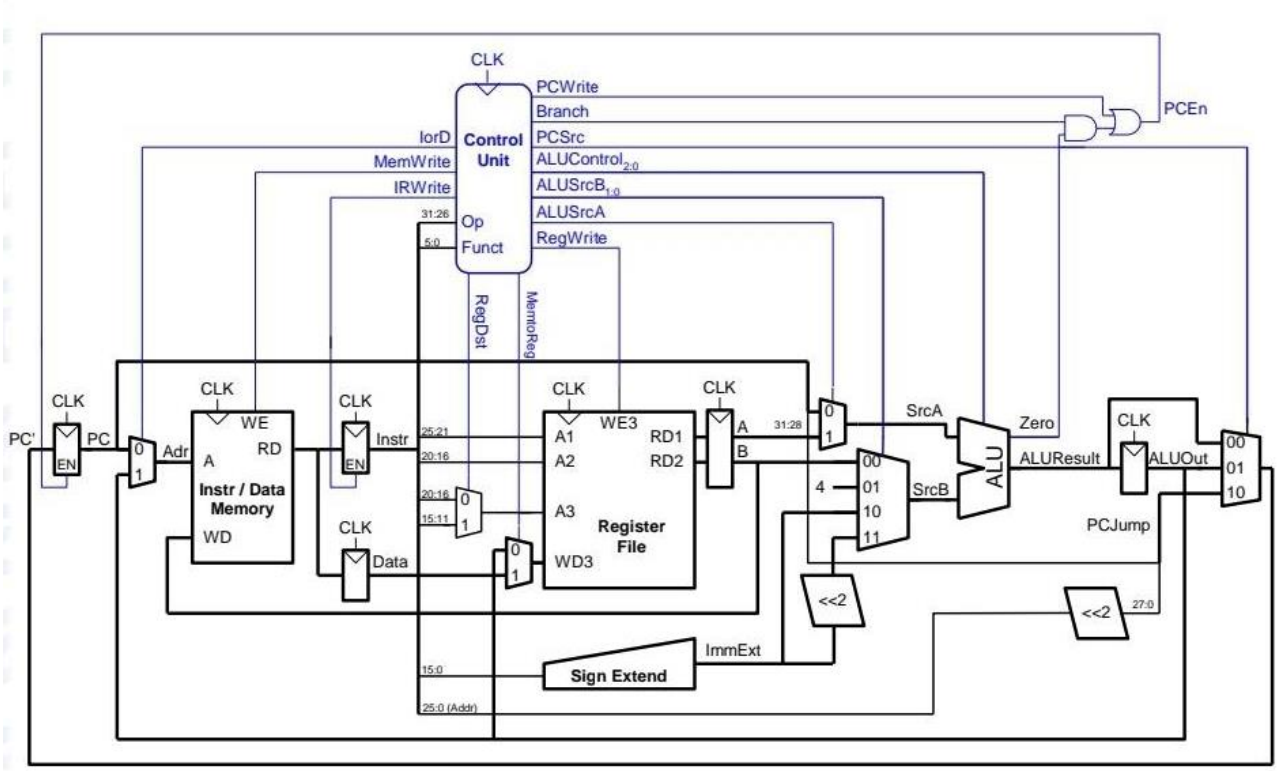
مرحله بعد مرحله Executing است که در آن بر اساس نوع دستورالعمل، محاسبات و اعمال منطقی در ALU انجام می‌شوند و خروجی‌ها برای نوشته شدن در حافظه یا رجیسترها و یا تغییر PC آماده می‌شوند. در صورتی که دستورالعمل‌ها از نوع Load/Store باشند پس از این مرحله وارد مرحله دیگری به نام Memory می‌شوند، و در غیر این صورت مرحله آخر یا Write Back اجرا می‌شود که در آن، مقادیری که حاصل نتیجه محاسبات ALU هستند و یا از حافظه خوانده شده‌اند، بر روی رجیسترهای مورد نظر نوشته می‌شوند و رجیسترفایل به روز می‌شود.

برخلاف پردازنده‌های تک سیکلی، در پردازنده‌های چند سیکلی هر یک از مراحل بالا می‌تواند در تعداد مشخصی سیکل ساعت انجام شود که این مسئله باعث می‌شود چنین پردازنده‌ای نسبت به پردازنده‌های تک سیکلی بتوانند با فرکانس کلاک بالاتری کار کنند و در مراحلی که زمان انجام آنها کوتاه‌تر است، وقفه اضافی نداشته باشیم. تفاوت دیگر این دو ساختار در آن است که در پردازنده‌های چند سیکلی، تنها یک واحد ALU و یک حافظه مشترک هم برای دستورالعمل‌ها و هم برای دیتا در نظر گرفته می‌شود، که به این

-

ترتیب از نظر منابع بهینه‌تر خواهد بود اما به سیگنال‌های کنترلی بیشتری نیاز خواهیم داشت و پیاده‌سازی مدار پیچیده‌تر خواهد بود. همچنین باید تعدادی رجیستر به مسیر داده اضافه کنیم تا در سیکل‌های مختلف کلاک، اطلاعات در آنها ذخیره شود.

تصویر زیر، شماتیک یک پردازنده چندسیکلی را نمایش می‌دهد، که در ادامه گزارش به توضیح هر یک از بخش‌های آن می‌پردازیم:



- **واحد کنترلی (Control Unit)**

یک واحد کنترلی بعد از خواندن هر دستورالعمل، عملیاتی که باید انجام شود را تعیین می کند و مدارهای مناسب را فعال می کند و نحوه جریان داده ها را در مسیر داده مشخص می کند.

وظیفه اصلی این بخش، تولید مجموعه‌ای از سیگنال‌های کنترلی است که به کمک آنها نحوه عملکرد سایر اجزای مدار تعیین خواهد شد. به عنوان مثال، پایه سلکت تمامی مالتی‌پلکسرها یک سیگنال کنترلی محسوب می‌شود که ورودی مالتی‌پلکسر را بر اساس مرحله‌ای که پردازنده در آن قرار گرفته است و نوع دستوری که در حال اجراست تعیین می‌کند.

در ادامه، به تفکیک هر یک از این سیگنال‌های کنترلی و نقش آنها را بررسی می‌کنیم:

- **PCSrc** : تعیین می‌کند که مقدار جدید PC برابر با مقدار قبلی PC بعلاوه 4 (برای رفتن به خط بعدی برنامه در حالت عادی) باشد، یا این مقدار جدید را خروجی ALU تعیین کند (برای دستوراتی مثل Branch و JR)

-

- **Branch**: در صورتی که دستورالعمل اجرا شده از نوع Branch باشد، این سیگنال 1 می‌شود تا در مرحله بعد مقدار ذخیره شده در رجیستر PC تغییر کند.
- **PCWrite**: در صورتی مقدار آن 1 خواهد شد که نتیجه دستور اجرا شده، مقدار PC را دستخوش تغییر کند. (برای دستورالعمل‌های J, JAL, JR, JALR استفاده می‌شود.)
- **IorD**: با توجه به اینکه در این ساختار از یک حافظه واحد برای دیتا و دستورالعمل استفاده می‌شود، به کمک این سیگنال تعیین می‌کنیم که محتوای نوشته‌شده/خوانده‌شده در حافظه از کدام نوع است.
- **MemWrite**: در صورتی که مقدار این سیگنال برابر با 1 باشد، مقدار جدیدی را در حافظه می‌نویسیم. در غیر این صورت چیزی در حافظه نوشته نشده و صرفا از آدرسی که ورودی به آن اشاره می‌کند دیتا یا دستورالعملی را می‌خوانیم.
- **MemtoReg**: مقدار این سیگنال تعیین می‌کند که Datapath از طریق خروجی ALU ادامه پیدا کند و یا از طریق خروجی حافظه (برای دستور Load)
- **IRWrite**: با 1 شدن این سیگنال، دستور بعدی Fetch می‌شود.
- **ALUSrcA**: تعیین می‌کند که ورودی اول ALU برابر با PC باشد و یا مقدار یک رجیستر.
- **ALUSrcB**: تعیین می‌کند که ورودی دوم ALU کدام یک از مقادیر زیر باشد: مقدار ذخیره شده در یک رجیستر، مقدار imm که از دستورالعمل استخراج شده و sign extend شده است، شماره خط دستور که تنها در دستورات خانواده Branch اتفاق می‌افتد، و یا عدد ثابت 4 (برای فرستادن PC به آدرس خط بعد.)
- **RegWrite**: در صورت 1 بودن مقدار این سیگنال، مقدار جدیدی در رجیسترفایل نوشته خواهد شد.
- **ALUControl**: با توجه به Opcode و Function دریافت شده در ورودی، تعیین می‌کند که دستورالعمل درحال پردازش از چه نوعی است و ALU باید چه عملیات منطقی یا ریاضی را بر روی ورودی‌ها انجام دهد.
- **RegDst**: تعیین می‌کند که رجیستر مقصد توسط کدام بخش از دستورالعمل ساخته می‌شود (در دستورالعمل‌های I-Type و R-Type متفاوت است.)

جدول مقادیر هر سیگنال کنترلی، به ازای دستورالعمل های خواسته شده:

		IorD	MemWrite	Branch	PcSrc	ALUOp	ALUScrA	ALUScrB	RegWrite	RegDst	MemtoReg	PcWrite
2	add	0	0	0	00	0000	1	01	1	0	0	0
3	sub	0	0	0	00	0001	1	01	1	0	0	0
4	addu	0	0	0	00	0000	1	01	1	0	0	0
5	subu	0	0	0	00	0001	1	01	1	0	0	0
6	and	0	0	0	00	0010	1	01	1	0	0	0
7	or	0	0	0	00	0011	1	01	1	0	0	0
8	xor	0	0	0	00	0100	1	01	1	0	0	0
9	nor	0	0	0	00	0101	1	01	1	0	0	0
10	slt	0	0	0	00	1011	1	01	1	0	0	0
11	sltu	0	0	0	00	1011	1	01	1	0	0	0
12	jr	0	0	0	00	0110	1	01	0	0	0	1
13	jalr	0	0	0	00	0110	1	01	1	0	0	1
14	multu	0	0	0	00	1000	1	01	1	0	0	0
15	mfhi	0	0	0	00	1001	1	01	1	0	0	0
16	mflo	0	0	0	00	1010	1	01	1	0	0	0
17	beq	0	0	1	10	1100	1	11	0	1	0	0
18	bne	0	0	1	10	1101	1	11	0	1	0	0
19	lw	1	0	0	00	0000	1	10	1	1	1	0
20	sw	1	1	0	00	0000	1	10	0	1	0	0
21	addi	0	0	0	00	0000	1	10	1	1	0	0
22	addiu	0	0	0	00	0000	1	10	1	1	0	0
23	slti	0	0	0	00	0101	1	10	1	1	0	0
24	sltiu	0	0	0	00	0101	1	10	1	1	0	0
25	andi	0	0	0	00	0010	1	10	1	1	0	0
26	ori	0	0	0	00	0011	1	10	1	1	0	0
27	xori	0	0	0	00	0100	1	10	1	1	0	0
28	lui	0	0	0	00	0111	1	10	1	1	0	0
29	J	0	0	0	01	-	-	-	0	-	-	1
30	Jal	0	0	0	01	-	-	-	1	-	-	1

-

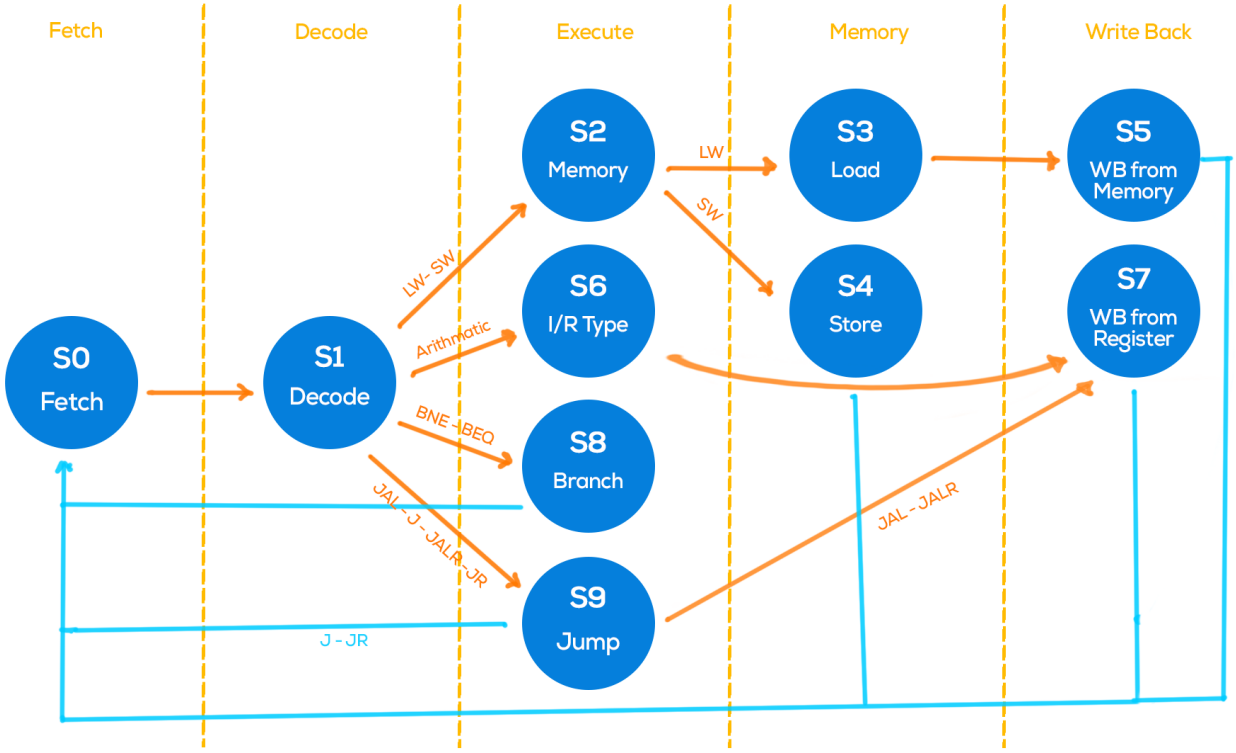
2-برای پیاده سازی، باید دستورات به صورت مناسب تقسیم بندی شوند و عملیاتی که در هر پالس ساعت باید انجام شود، مشخص گردند .دستوراتی که باید پیاده سازی شوند شامل دوبخش اند: اوّل دستورات ساده تری که در درس معماری کامپیوتر شبیه آن در پردازنده V-RISC توضیح داده شده بود و دوّم دستورات تکمیلی که لازم است در این آزمایش به دسته اول اضافه شوند و در زیر با فونت پررنگ مشخص شده اند:

R format: add, sub, addu, subu, and, or, xor, nor, slt, sltu, **jr, jalr, multu, mfhi, mflo**

I format: beq, bne, lw, sw, addi, addiu, slti, sltiu, andi, ori, xori, **lui**

J format: j, jal

یکی از مواردی که تقسیم‌بندی دستورالعمل‌ها و تفکیک عملکرد واحد کنترلی را مشخص می‌کند، Finite State Machine مربوط به حالات مختلف عملکرد پردازنده است. بنابراین در همین بخش به توضیح FSM پیاده‌سازی شده نیز می‌پردازیم:



همانطور که ملاحظه می‌شود، بعضی دستورالعمل‌ها برای اجرا نیاز به طی کردن هر پنج مرحله (که پیش از این توضیح داده شدند) دارند، در صورتی که برخی دستورالعمل‌ها تنها از 3 یا 4 مرحله عبور می‌کنند. بنابراین برای بهینه‌سازی زمان‌بندی اجرای دستورات می‌توان از این ایده استفاده کرده و دستورالعمل‌هایی که تعداد مراحل برابری دارند را در یک دسته قرار داد.

بر همین اساس، 10 حالت (State) مختلف برای سیستم تعریف شده است که ارتباط بین هر یک از آنها در نمودار بالا قابل ملاحظه است. در هر کلاک، با توجه به دستورالعمل درحال پردازش سیستم وارد حالت بعدی شده و با رسیدن به انتهای مسیر به حالت نخست که همان مرحله Fetching است بازمی‌گردد.

-

تعیین state در هر لحظه و همچنین تشخیص state بعدی، یکی از وظایف اصلی واحد کنترلی است. پس از آن، با توجه به حالتی که سیستم در آن قرار گرفته است، مقدار هر یک سیگنال‌های کنترلی مشخص شده و به مسیر داده برگردانده می‌شود.

به این ترتیب، پیاده‌سازی کد مربوط به واحد کنترلی شامل بخش‌های زیر خواهد بود:

1. تعریف state ها به صورت type، و تعریف سیگنال مربوط به حالت فعلی و حالت بعدی

```
-------------------------------------current_s ------------------------------------------------
   type current_s_type is (Fetch, Decode, Memory, RItype, Branching, Jump, Load, Store, WBfromMem, WBfromReg );

   -- current_s Signals
   signal current_s, next_s: current_s_type;

begin
```

2. تغییر حالت به حالت بعدی، در هر لبه بالارونده کلاک

```
---------------------------------- current state assignment ---------------------------------------
fsm_assignment: process (clk) is
   begin
      if rising_edge(clk) then
            current_s <= next_s;
      end if;
end process fsm_assignment;
```

3. پیاده سازی FSM

```
-------------------------------- Finite state Machine ----------------------------------
finite_state_machine: process (current_s, Op) is
begin
    case current_s is

        when Fetch =>
           next_s <= Decode;

        ------------------------------------------------------------------------------

        when Decode =>
           if Op(5) = '1' then
           -- Load and Store Instructions
              next_s <= Memory;

           elsif Op = "000010" or Op = "000011" then
           --J-Type instructions
              next_s <= Jump;

           elsif Op = "000000" or Op(3) = '1' then
           -- Arithmatic instructions
              next_s <= RItype;

           elsif Op = "000100" or Op = "000101" then
           -- Branch instructions
              next_s <= Branching;

           else
           -- No Operarion
           next_s <= Fetch;


           end if;
        ------------------------------------------------------------------------------

        when Memory =>
           if Op = "100011" then
              next_s <= Load;

           elsif Op = "101011" then
              next_s <= Store;

           end if;
        ------------------------------------------------------------------------------

        when RItype =>
              next_s <= WBfromReg;
```

-

4. تعیین مقادیر سیگنال‌های کنترلی با توجه به حالت سیستم

```
------------------------+----- PC-Related Control Signals ---------------------------------

PCWrite <=          '1' when current_s = Fetch else
                    '1' when current_s = RItype and Funct = "001000" else
                    '1' when current_s = RItype and Funct = "001001" else
                    '1' when current_s = Jump else '0';

PcSrc <=            "10" when current_s = Jump else
                    "00" when current_s = Branching else "01";

------------------------------Other Control Signals: ----------------------------------

IorD     <=         '1' when current_s = Load else
                    '1' when current_s = Store else '0';
-------------------------------------------------------------------------
MemWrite <=         '1' when current_s = Store else '0';
-------------------------------------------------------------------------
Branch   <=         '1' when current_s = Branching else '0';
-------------------------------------------------------------------------
MemtoReg <=         '1' when current_s = Load else '0';
-------------------------------------------------------------------------
IRWrite  <=         '1' when current_s = Fetch else '0';
-------------------------------------------------------------------------
RegDst   <=         '0' when current_s = Branching else
                    '0' when current_s = RItype and Op(3) = '1' else '1';
-------------------------------------------------------------------------
RegWrite <=         '0' when current_s = Branching else
                    '0' when current_s = Store else
                    '0' when current_s = Jump and Op = "000010" else
                    '0' when current_s = RItype and Funct = "001000" else '1';
-------------------------------------------------------------------------
ALUSrcA  <=         '0' when current_s = Fetch else '1';
------------------------------------------------------------------------ !BEPORS - BEPORS!
ALUSrcB  <=         "10" when current_s = RItype and Op(3) = '1' else
                    "11" when current_s = Branching else
                    "01" when current_s = Fetch  else "00";
-------------------------------------------------------------------------
ALUCtrl    <=       "0000" when current_s = RItype      and  (Funct(5 downto 1) = "10000" or Op(5 downto 1) = "00100") else  --add, addu, addi, addui
                    "0000" when current_s = Memory      or current_s = Store or current_s = Load else -- lw, sw
                    "0001" when current_s = RItype      and  Funct(5 downto 1) = "10001" else --sub, subu
                    "0010" when current_s = RItype      and (Funct = "100100" or Op = "001100") else --and, andi
                    "0011" when current_s = RItype      and (Funct = "100101" or Op = "001101") else --or, ori
                    "0100" when current_s = RItype      and (Funct = "100110" or Op = "001110") else --xor, xori
                    "0101" when current_s = RItype      and  Funct = "100111" else --nor
                    "0000" when current_s = RItype      and (Funct(5 downto 1) = "10101" or Op(5 downto 1) = "00101") else --slt, sltu, slti, sltiu
                    "1000" when current_s = RItype      and Funct = "011001" else --multu
                    "1001" when current_s = RItype      and Funct = "010000" else --mfhi
                    "1010" when current_s = RItype      and Funct = "010010" else --mflo
                    "1100" when current_s = Branching   and Op = "000100" else --BEQ
                    "1101" when current_s = Branching   and Op = "000101" else --BNE
                    "0111" when current_s = RItype and  Op = "001111" else --lui
                    "0110" when current_s = RItype and  Op = "001111" else "1111"; --jr, jalr
-------------------------------------------------------------------------
```

توضیحات مربوط به جزییات کدنویسی، در ویدیوی ضمیمه‌شده ارائه خواهد شد.

-

- **مسیر داده (Data Path)**

همانگونه که پیش‌تر توضیح داده شد، مسیر داده شامل اجزای گوناگونی است که در ادامه نقش هریک در مدار به تفکیک توضیح داده شده و کد مربوط به پیاده‌سازی هریک بررسی خواهد شد.

- **مالتی پلکسرها :**

مجموعا 4 مالتی‌پلکسر استفاده شده است:

مالتی پلکسر 2 به 1 اول، تعیین می‌کند که از حافظه مشترک دستورالعمل و دیتا، کدام یک خوانده خواهد شد. این مالتی‌پلکسر از سیگنال IorD به عنوان پایه سلکت استفاده می‌کند.

مالتی پلکسر 2 به 1 دوم تعیین می‌کند که آدرسی که به یک رجیستر اشاره می‌کند توسط خروجی ALU تعیین شده است و یا از حافظه خوانده شده است. این مالتی‌پلکسر از سیگنال MemtoReg به عنوان پایه سلکت استفاده می‌کند.

مالتی پلکسر 3 به 1، تعین می‌کند که مقداری که در رجیستر PC نوشته می‌شود، از چه منبعی باشد: PC+4، مقدار آدرس مربوط به دستورالعمل های Jump، ویا مقدار حاصل از شیفت دادن imm در دستورالعمل‌های خانواده Branch. این مالتی‌پلکسر با سیگنال PCSrc کنترل می‌شود.

مالتی پلکسر 4 به 1 تعیین می‌کند که ورودی دوم به ALU از چه منبعی باشد: رجیستر (مربوط به دستورات R-type)، مقدار ثابت imm (مربوط به دستورات I-type)، عدد ثابت 4 (برای فرستادن PC به خط بعد دستورات) و یا مقدار شیفت‌یافته imm که مربوط به دستورات Branch است. این مالتی‌پلکسر از سیگنال ALUSrcB به عنوان پایه سلکت استفاده می‌کند.

نمونه کد پیاده‌سازی مالتی‌پلکسر:

```
entity PCMUX is
    Port ( ALUResult : in  STD_LOGIC_VECTOR (31 downto 0);
           ALUOut : in  STD_LOGIC_VECTOR (31 downto 0);
           PCSrc : in  STD_LOGIC_VECTOR (1 downto 0);
           PC : in  STD_LOGIC_VECTOR (31 downto 0);
           Addr : in  STD_LOGIC_VECTOR (25 downto 0);
           PCin : out  STD_LOGIC_VECTOR (31 downto 0));
end PCMUX;

architecture Behavioral of PCMUX is

    signal PCJump : STD_LOGIC_VECTOR (31 downto 0);

begin

   PCJump <= PC(31 downto 28) & Addr & "00";

   with PCSrc select PCin <=
          ALUResult when "00",
          ALUOut    when "01",
          PCJump    when others;

end Behavioral;
```

-

- **رجیسترها:**

همانگونه که در توضیحات ابتدای گزارش اشاره شد، در ساختار پردازنده‌های چند سیکلی، با توجه به اینکه دستورالعمل‌ها در یک سیکل ساعت انجام نمی‌شوند و با فرا رسیدن لبه بالارونده بعدی کلاک نباید اطلاعات کدگشایی شده و درحال پردازش از بین برود، از تعدادی رجیستر برای نگهداری این اطلاعات استفاده می‌شود. در ساختار پردازنده پیاده شده، مجموعا 5 رجیستر استفاده شده است:

از رجیستر اول برای نگهداری مقدار PC استفاده می‌شود. پایه En این رجیستر مطابق زیر تعیین می‌شود:

PCWrite *OR* (Branch *AND* Zero)

به این ترتیب در دو صورت مقدار موجود در این رجیستر به روز می‌شود: اول، در صورتی که یکی از دستورات خانواده Jump اجرا شده باشد، و دوم، وقتی یکی از دستورات خانواده Branch اجرا شده و شرط آن برقرار بوده باشد. در غیر این صورت، با رسیدن دوباره به حالت Fetch مقدار مربوط به خط بعد دستورالعمل‌ها در این رجیستر ریخته می‌شود.

از رجیستر دوم برای نگهداشتن دستورالعمل درحال پردازش تا زمان Fetch شدن دستورالعمل بعدی استفاده می‌شود. با 1 شدن سیگنال IRWrite، مقدار این رجیستر به روز می‌شود.

از رجیستر سوم برای نگهداری دیتای خوانده شده از حافظه استفاده می شود. این رجیستر با لبه بالارونده کلاک کنترل می‌شود.

از رجیستر چهارم برای نگهداری مقادیری استفاده می‌شود که توسط دستورات R-type و I-type از رجیسترفایل خوانده شده‌اند. این رجیستر نیز با لبه بالارونده کلاک کنترل می‌شود.

از رجیستر پنجم برای نگهداری مقدار محاسبه‌شده خروجی ALU استفاده می‌شود که با رسیدن لبه بالارونده مقدار آن به روز می‌شود.

نمونه کد پیاده سازی رجیستر:

```
architecture Behavioral of ProgCnt is

begin

   process (clk, PCEn)
   begin

      if (clk'event and clk = '1') then

         if(PCEn = '1') then

            PC <= PCin;

         end if;

      end if;

   end process;

end Behavioral;
```

-

○ Sign Extend

این بخش مربوط به دستوراتی است که در آن‌ها مقدار immediate تعریف می‌شود. هنگامی که بخشی از 32 بیت مربوط به Instruction عدد ثابت باشد، چون این مقدار تنها 16 بیت از هر instruction خواهد بود، لازم است با تکرار بیت علامت آن، به 32 بیت برسد تا در صورت نیاز به عنوان operand دوم ورودی ALU استفاده شود. کد زیر این عملیات را انجام می‌دهد:

```
35  entity SgnExt is
36      Port ( imm16 : in  STD_LOGIC_VECTOR (15 downto 0);
37             imm32 : out  STD_LOGIC_VECTOR (31 downto 0));
38  end SgnExt;
39
40  architecture Behavioral of SgnExt is
41
42  begin
43
44     imm32 <=  (x"ffff" & imm16) when (imm16(15)='1') else  (x"0000" & imm16);
45
46  end Behavioral;
```

○ Memory

در این بخش یک آرایه 32x32 به عنوان memory تعریف شده که خانه‌های ابتدای آن بخش Instructionmemory را تشکیل می‌دهند و به همین دلیل مقداردهی اولیه شده اند. این مقداردهی وابسته به کد اسمبلی بوده که ما برای نمونه می‌خواستیم اجرا کنیم. هر خط کد را به 32 بیت دستور تبدیل کرده و مقدار هگز آن را به عنوان مقدار اولیه مموری دادیم. مابقی خانه ها با صفر مقداردهی شده اند تا نقش DataMemory را داشته باشند. یک ورودی این ماژول Adr است که بنا به انتخاب سیگنال کنترلیIorD، مقدار PC یا ALUResult را به خود می‌گیرد و سپس بنا به تصمیم سیگنال کنترلی MemWrite، مقداری که در خانه شماره Adr ام حافظه ذخیره شده را در رجیستر RD یا همان ReadData می‌ریزد و یا مقدار رجیستر WD یا همان WriteData را در خانه شماره Adr ام حافظه ذخیره خواهد کرد. تکه کد زیر این ماژول را ساخته است:

```
44
45      type Mem_type is array (0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
46      Signal mem : Mem_type := (
47         x"3c010ff0",    -- LUI  R1, 0x0FF0
48         x"3c010008",    -- ADDI R2, R0, 0x0008
49         x"14220005",    -- BNE  R1, R2, branch
50         x"ac610000",    -- SW   R3, 0(R1)
51         x"00221822",    -- branch: SUB  R3, R1, R2
52         x"ac420000",    -- SW   R1, 0(R2)
53         others => x"00000000");
54
55  begin
56
57      process (clk, MemWrite)
58      begin
59         if (clk' event and clk = '1') then
60            if (MemWrite = '1') then
61               mem(conv_integer(Adr)) <= B;
62            elsif (MemWrite = '0') then
63               RD <= mem(conv_integer(Adr));
64            end if;
65         end if;
66      end process;
67
68  end Behavioral;
```

-

- **Register File**

این ماژول هم یک آرایه 32x32 دارد که رجیسترهای32 بیتی 0 تا 32 را نشان می‌دهد و همگی با صفر مقداردهی شده‌اند. در این بخش هرگاه کلاک بیاید، در صورتی که سیگنال کنترلی RegWrite یک باشد، مقدار WriteData که بسته به انتخاب MemtoReg خروجی ALU یا دیتای خروجی مموری است، در رجیستر شماره Writereg ذخیره خواهد شد و اگر سیگنال کنترلی RegWrite صفر باشد، مقدار ذخیره شده در رجیسترهای شماره readreg1,2 به ترتیب در خروجی readdata1,2 ذخیره خواهند شد. این مقادیر پس از رجیستر فایل، در یک رجیستر نگه داشته می‌شوند تا در هنگام لزوم، به عنوان دو ورودی ALU استفاده شوند. این ماژول به صورت زیر نوشته شده است:

باید توجه داشت که رجیستر R_0 همواره برای مقدار صفر رزرو شده و حتی اگر کاربر بخواهد به آن مقدار غیر صفر بدهد، این امکان از او گرفته شده است.

```
48    type RAM_type is array(0 to 31) of std_logic_vector(31 downto 0);
49    signal ram : RAM_type := (x"00000000", x"00000000", x"00000000", others => x"00000000");
50
51 begin
52
53    process (clk)
54    begin
55       if (clk'event and clk= '1') then
56
57             if (RegWrite = '1') then
58                ram(conv_integer(writereg)) <= writedata;
59             end if;
60
61             if (conv_integer(readreg1) = 0) then
62                readdata1 <= x"00000000";
63             else
64                readdata1 <= ram(conv_integer(readreg1));
65             end if;
66
67             if (conv_integer(readreg2) = 0) then
68                readdata2 <= x"00000000";
69             else
70                readdata2 <= ram(conv_integer(readreg2));
71             end if;
72       end if;
```

- **ALU**

در این بخش دو Operand ورودی داریم که توسط مالتی پلکسرهای قبل از این ماژول مقدارشان تعیین شده. روی این دو مقدار بسته به مقدار سیگنال کنترلی ALUCtrl عملیات مختلفی انجام می‌شود. برای مثال در صورت آمدن سیگنال "0000"، بسته به اینکه Operand دوم چه باشد، جمع عادی یا addi صورت می‌گیرد. توجه داریم که در صورت سوال ذکر شده از تولید exception در عملیات جمع و تفریق اعداد عالمتدار صرفنظر کنیم، پس جمع و تفریق علامتدار و بی‌علامت مشابه خواهند شد.

عملیات جمع، تفریق، and، or، xor، nor هم به همین صورت روی دو ورودی ALU انجام می‌شوند. سپس برای دستورات slt/sltu/slti/sltiu مقدار دو ورودی با هم مقایسه می‌شود و اگر اولی کوچکتر بود، خروجی یک و در غیر این صورت صفر خواهد بود. ضرب هم در این واحد انجام شده و چون حاصل 64 بیتی دارد، بسته به دستور بعد از آن که mfhi یا mflo باشد، 32 بیت پرارزش یا کم ارزش را در خروجی نمایش می‌دهد.

-

برای دستورات beq و bne هم برابری دو ورودی بررسی شده و برروی flag صفر(zero) اثر می‌گذارد.

محاسبات مربوط به دستورات lui/ jr/jalr هم در این واحد به صورت زیر انجام می‌شود:

```
50
51    Process (SrcA, SrcB, ALUctrl)
52    begin
53       case (ALUctrl) is
54
55       -- Arithmatic-Unit
56       when "0000" => ALUResult <= SrcA + SrcB;
57                      zero  <= '0';
58       when "0001" => ALUResult <= SrcA - SrcB;
59                      zero  <= '0';
60
61       -- Logical-Unit
62       when "0010" => ALUResult <= SrcA and SrcB;
63       when "0011" => ALUResult <= SrcA or SrcB;
64       when "0100" => ALUResult <= SrcA xor SrcB;
65       when "0101" => ALUResult <= not(SrcA or SrcB);
66
67       when "1011" => -- slt/sltu/slti/sltiu
68          if (SrcA < SrcB) then ALUResult <= "00000000000000000000000000000001";
69          elsif (SrcA > SrcB) then ALUResult <= "00000000000000000000000000000000";
70          end if;
71
72       when "1000" => -- multiply
73          mulRes <= SrcA * SrcB;
74       when "1001" => -- mfhi
75          ALUResult <= mulRes(63 downto 32);
76       when "1010" => -- mflo
77          ALUResult <= mulRes(31 downto 0);
78
79       when "1100" => -- beq
80          if (SrcA = SrcB) then zero <= '1';
81          else zero <= '0';
82          end if;
83       when "1101" => -- bne
84          if (SrcA = SrcB) then zero <= '0';
85          else zero <= '1';
86          end if;
87
88       when "0111" => -- lui
89          ALUResult <= SrcB(15 downto 0) & "0000000000000000";
90
91       when "0110" => -- jr/jalr
92          ALUResult <= SrcA(29 downto 0) & "00";
93
94       when others => ALUResult <= (others => '0');
95                      zero <= '0';
96
97       end case;
98
99       end process;
```

-

توضیحات مربوط به کد Assembly نوشته شده

از کد اسمبلی زیر برای تست کردن پردازنده پیاده‌سازی شده استفاده شده است:

```
LUI  R1, 0x0FF0
ADDI R2, R1, 0x0008
BNE  R1, R2, branch
SW   R3, 0(R1)
branch:
SUB  R3, R1, R2
SW   R1, 0(R2)
```

با کمک این برنامه، ابتدا مقدار 0x0ff0 در 16 بیت بالای رجیستر R1 ذخیره می‌شود. سپس این مقدار با مقدار ثابت 0x0f0f جمع شده و حاصل آن در رجیستر R2 ذخیره خواهد شد. در ادامه مقادیر موجود در رجیسترهای R1 وR2 با یکدیگر مقایسه می‌شوند و با توجه به برابر بودن، برنامه به لیبل Branch خواهد رفت. به این ترتیب ابتدا مقدار موجود در رجیستر R2 از مقدار موجود در رجیستر R1 کم می‌شود و نتیجه حاصل در R3 ذخیره می‌شود، و سپس مقدار موجود در رجیستر R1 در آدرسی که رجیستر R2 به آن اشاره می‌کند (در اینجا برابر با آدرس خانه هشتم حافظه) ذخیره می‌شود.

می‌دانیم فرمت دستورات در این معماری به شکل زیر تعریف می‌شود:

I-Format:

op	rs	rt	immediate	
001000	10011	01010	0000000000000100	Example: addi $t2, $s3, 4

J-Format:

op	address	
000010	00000000000000000100000001	Example: j LOOP (or j 1028)

R-Format:

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	Example: add $s0, $s1, $s2

بنابراین دستورات اسمبلی بالا، مطابق زیر به صفر و یک ترجمه شده و در حافظه ذخیره می‌شوند:

001111 00000 00001 0000 0000 0000 1000 ,0X3C010008

001000 00001 00010 0000 1111 0000 1111 ,0X20220F0F

000101 00001 00010 0000 0000 0000 0101 ,0X14220005

101011 00011 00001 0000 0000 0000 0000 ,0XAC610000

000000 00001 00010 00011 00000100010 ,0X00221822

101011 00001 00010 0000 0000 0000 0000 ,0XAC220000

-

توضیحات مربوط به کد Top Level:

در این فایل، ماژول هایی که در قسمت قبل یک به یک توضیح داده شدند به صورت component تعریف شده‌اند و ورودی‌ها و خروجی‌های هر یک به یکدیگر متصل است (از طریق port map).

به طور کلی در این قسمت تنها ورودی ما سیگنال کلاک است، و مراحل مختلف اجرای دستورات با لبه بالارونده کلاک سنکرون هستند. البته یک خروجی نیز صرفا برای تست اضافه شده‌است تا امکان تهیه تست‌بنچ از کد وجود داشته باشد.

ورودی و خروجی های سایر ماژول‌ها در فایل Top level به عنوان سیگنال تعریف می‌شود، که این سیگنال‌ها درواقع نقش سیم‌های رابط موجود در مسیر داده را ایفا می‌کنند.

همچنین گیت‌های مربوط به ساخت سیگنال PCEn در این لایه از کد پیاده‌سازی می‌شوند.

- نمونه تعریف سیگنال و component ها

```
signal Instr : std_logic_vector(31 downto 0);
signal RegDst : std_logic;
signal readreg1 : std_logic_vector(4 downto 0);
signal readreg2 : std_logic_vector(4 downto 0);
signal writereg : std_logic_vector(4 downto 0);
signal Op : std_logic_vector(5 downto 0);
signal Funct : std_logic_vector(5 downto 0);
signal imm16 : std_logic_vector(15 downto 0);
signal Addr : std_logic_vector(25 downto 0);
signal IRWrite : std_logic;

COMPONENT InstDec
PORT(
   Instr : IN std_logic_vector(31 downto 0);
   RegDst : IN std_logic;
   readreg1 : OUT std_logic_vector(4 downto 0);
   readreg2 : OUT std_logic_vector(4 downto 0);
   writereg : OUT std_logic_vector(4 downto 0);
   Op : OUT std_logic_vector(5 downto 0);
   Funct : OUT std_logic_vector(5 downto 0);
   imm16 : OUT std_logic_vector(15 downto 0);
   Addr : OUT std_logic_vector(25 downto 0)
   );
END COMPONENT;

COMPONENT InstReg
PORT(
   RD : IN std_logic_vector(31 downto 0);
   clk : IN std_logic;
   IRWrite : IN std_logic;
   Instr : OUT std_logic_vector(31 downto 0)
   );
END COMPONENT;
```

-

- اتصال ماژول ها با استفاده از Port map

```
Inst_CtrlUnit: CtrlUnit PORT MAP(
    Op => Op,
    Funct => Funct,
    clk => clk,
    PCSrc => PCSrc,
    PCWrite => PCWrite,
    RegDst => RegDst,
    Branch => Branch,
    MemtoReg => MemtoReg,
    ALUctrl => ALUctrl,
    MemWrite => MemWrite,
    IorD => IorD,
    IRWrite => IRWrite,
    ALUSrcA => ALUSrcA,
    ALUSrcB => ALUSrcB,
    RegWrite => RegWrite
);

Inst_ABreg: ABreg PORT MAP(
    readdata1 => readdata1,
    readdata2 => readdata2,
    clk => clk,
    A => A,
    B => B
);
```

- ساخت سیگنال PCEn:

```
PCEn <= PCwrite or (Branch and zero);
```

-

بررسی نتایج

در این بخش به بررسی صحت عملکرد هرکدام از ماژول‌های اصلی پرداختیم. پس برای هرکدام یک testbench مجزا طراحی شده و ورودی‌ها و خروجی‌ها نمایش داده شدند.

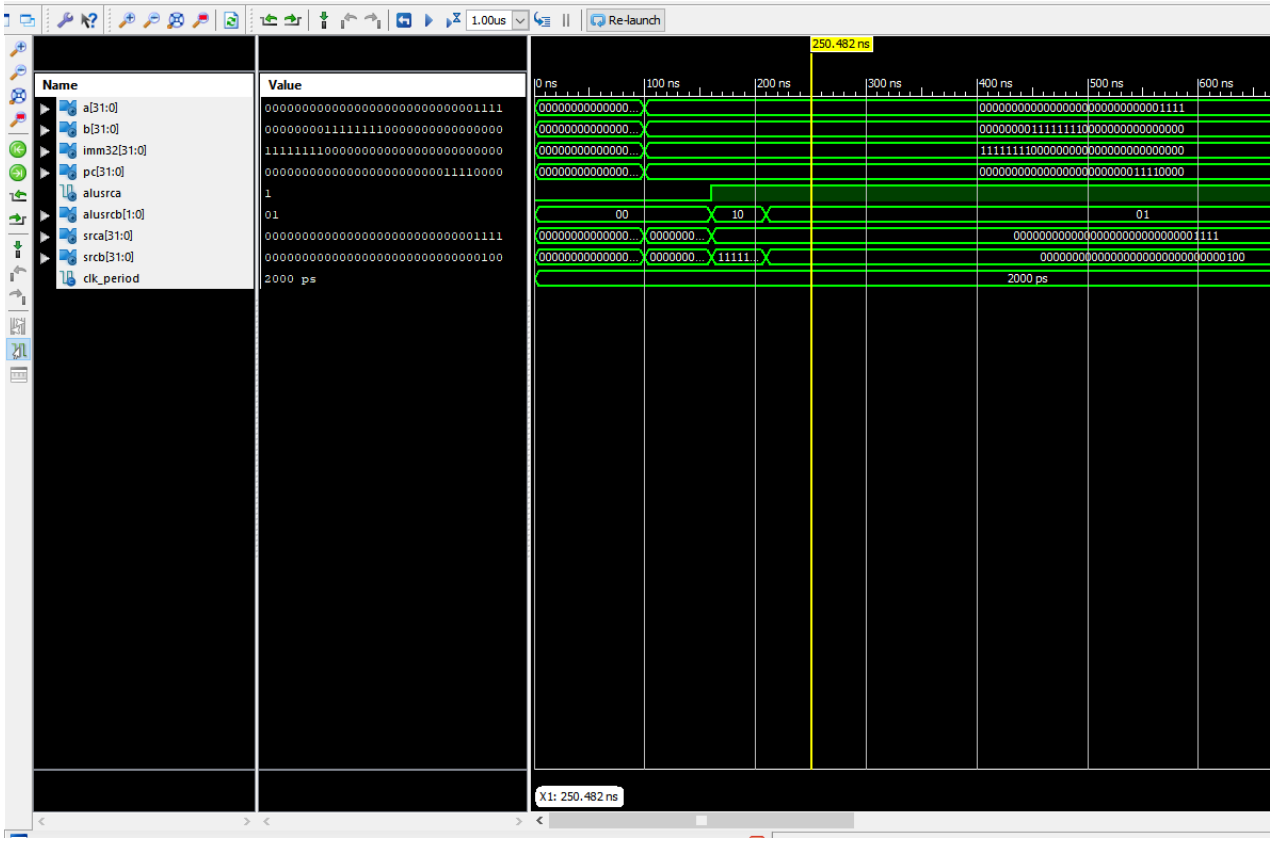
برای نمونه ورودی‌های مالتی پلکسر پیش از ALU را به شکل زیر مقداردهی کرده و با دادن مقادیر مختلف به سیگنال‌های کنترلی ALUSrcA,B در زمان‌های متفاوت، خروجی را مشاهده کردیم:

A <= x"0000000F;"

PC <= x"000000F0;"

imm32 <= x"FF000000;"

B <= x"00FF0000;"



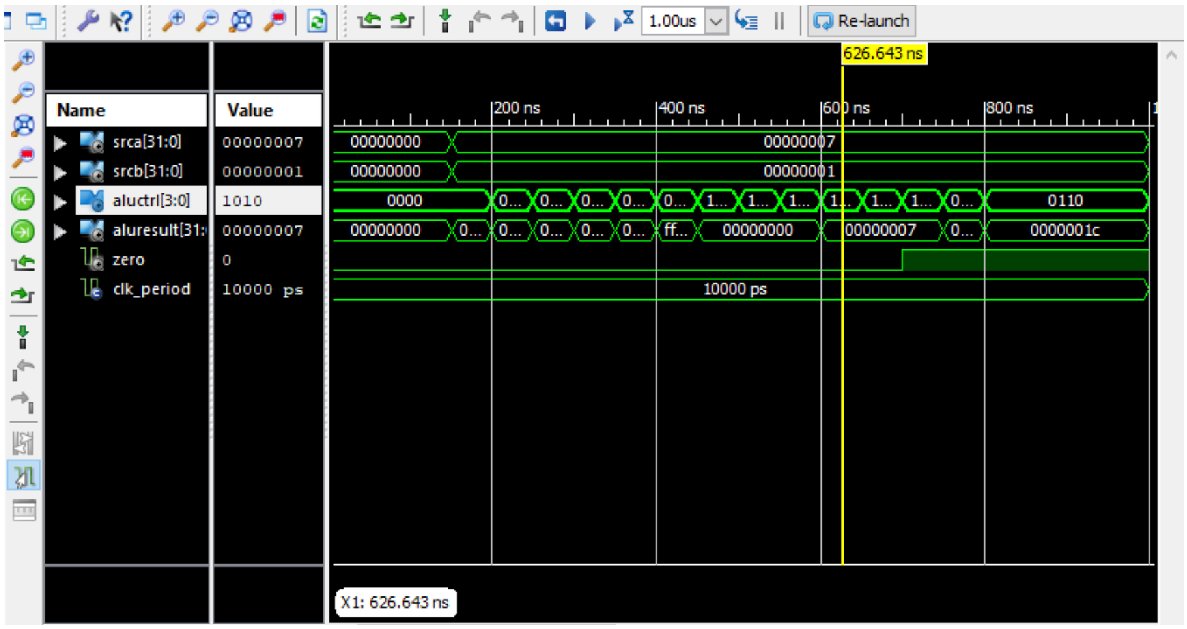
یا به ALU دو مقدار 7 و 1 را داده و با دادن مقادیر مختلف به ALUCtrl، نتیجه انجام عملیات مختلف را روی آن دو دیدیم:

مثلا در زمان نشان داده شده در شکل زیر، عملیات ضرب انجام شده و 32 بیت کم ارزش در خروجی نمایش داده شده که همان عدد 7 است: (در همین تصویر، صحیح بودن نتیجه سایر عملیات‌ها هم مشخص است.)

-

یک بار هم مقادیر زیر را با فواصل 20 نانو ثانیه‌ای به عنوان ورودی به کنترل یونیت داده و سیگنال‌های کنترلی را مشاهده کرده و دیدیم به درستی مقدار می‌گیرند:

--R-Type Test



Op <= "000000;"

Funct <= "100100;"

--Branch Test

Op <= "000100;"

Funct <= "000000;"

--Jump Test

Op <= "000010;"

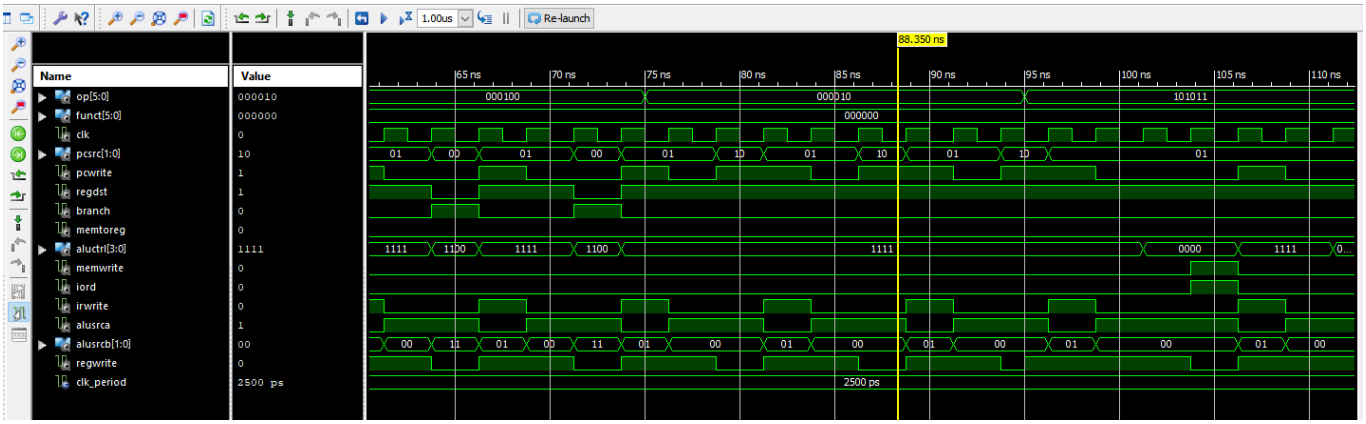
Funct <= "000000;"

--Load/Store Test

Op <= "101011;"

Funct <= "000000;"

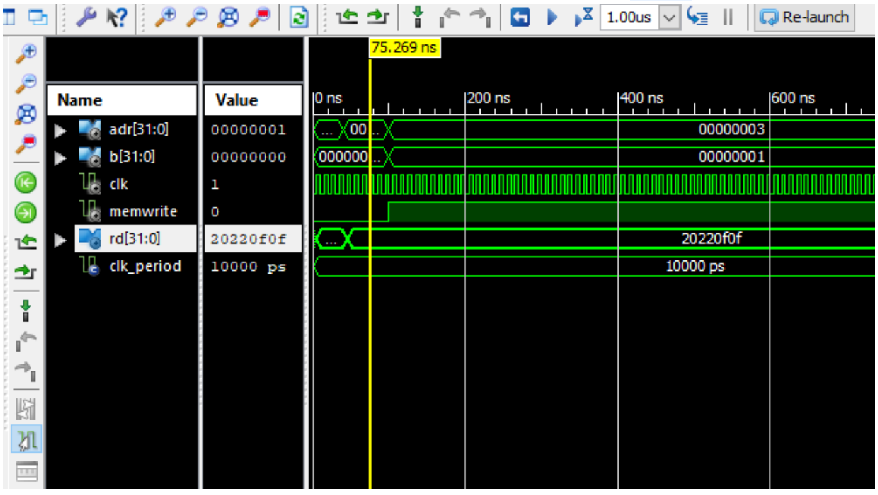
-



برای بررسی عملکرد صحیح مموری، مقادیر زیر را به عنوان ورودی داده و نتایج را در سیگنال خروجی و خانه‌های مموری مشاهده کردیم:

```
-- Read

MemWrite <= '0';

Adr <= x"00000001";

wait for 60 ns;


-- Write

MemWrite <= '1';

Adr <= x"00000003";

B <= x"00000001";

wait for 60 ns;
```

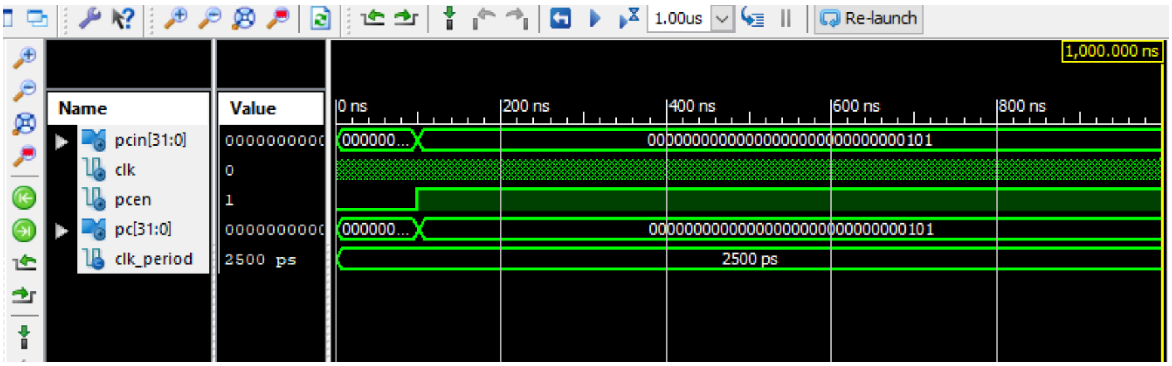
	0	1
0x0	3C010FF0	20220F0F
0x2	14220005	00000001
0x4	00221822	00000000
0x6	00000000	00000000
0x8	00000000	00000000
0xA	00000000	00000000
0xC	00000000	00000000
0xE	00000000	00000000
0x10	00000000	00000000
0x12	00000000	00000000
0x14	00000000	00000000
0x16	00000000	00000000
0x18	00000000	00000000
0x1A	00000000	00000000
0x1C	00000000	00000000
0x1E	00000000	00000000



مطابق انتظار، مقدار 0x00000001 در خانه 3 حافظه ذخیره شد و مقدار 0x20220f0 که در خانه 1 حافظه ذخیره شده بود روی رجیستر RD ریخته شد.

-

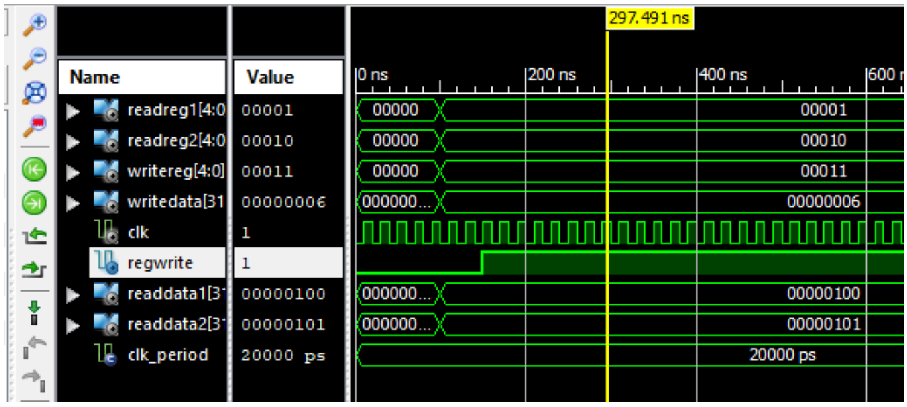
به ورودی ماژول PC مقدار 0x00000005 داده و دیدیم با آمدن کلاک و درصورت یک شدن سیگنال enable آن، این مقدار را به خروجی می‌دهد.



ورودی‌های رجیسترفایل را با این مقادیر تعیین کرده و نتایج را در رجیسترهای مختلف دیدیم:

```
readreg1 <= "00001";

readreg2 <= "00010";

writereg <= "00011";

writedata <= x"00000006";

RegWrite<= '0';

wait for 50 ns;

RegWrite <= '1';

wait for 50 ns;
```

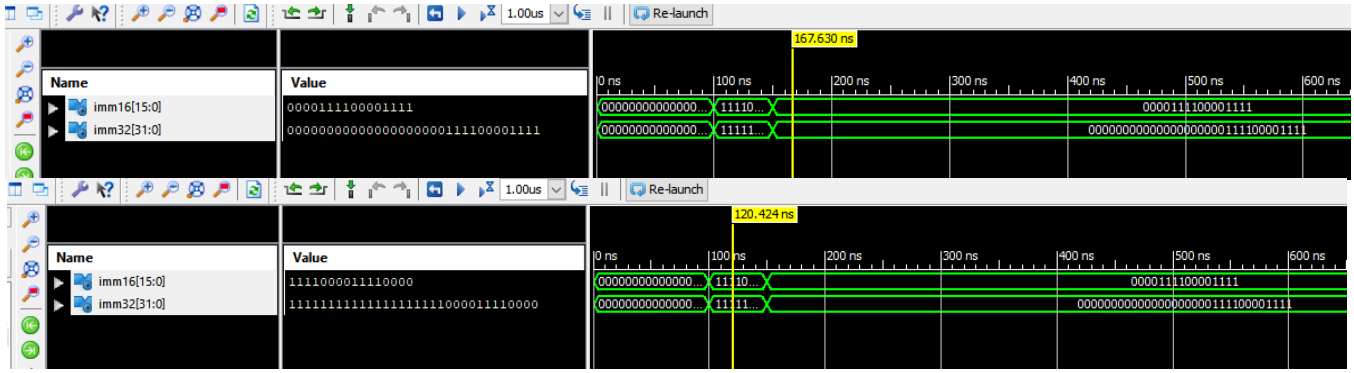
	0	1
0x0	00000000	00000100
0x2	00000101	00000006
0x4	00000000	00000000
0x6	00000000	00000000
0x8	00000000	00000000
0xA	00000000	00000000
0xC	00000000	00000000
0xE	00000000	00000000
0x10	00000000	00000000
0x12	00000000	00000000
0x14	00000000	00000000
0x16	00000000	00000000
0x18	00000000	00000000
0x1A	00000000	00000000
0x1C	00000000	00000000
0x1E	00000000	00000000



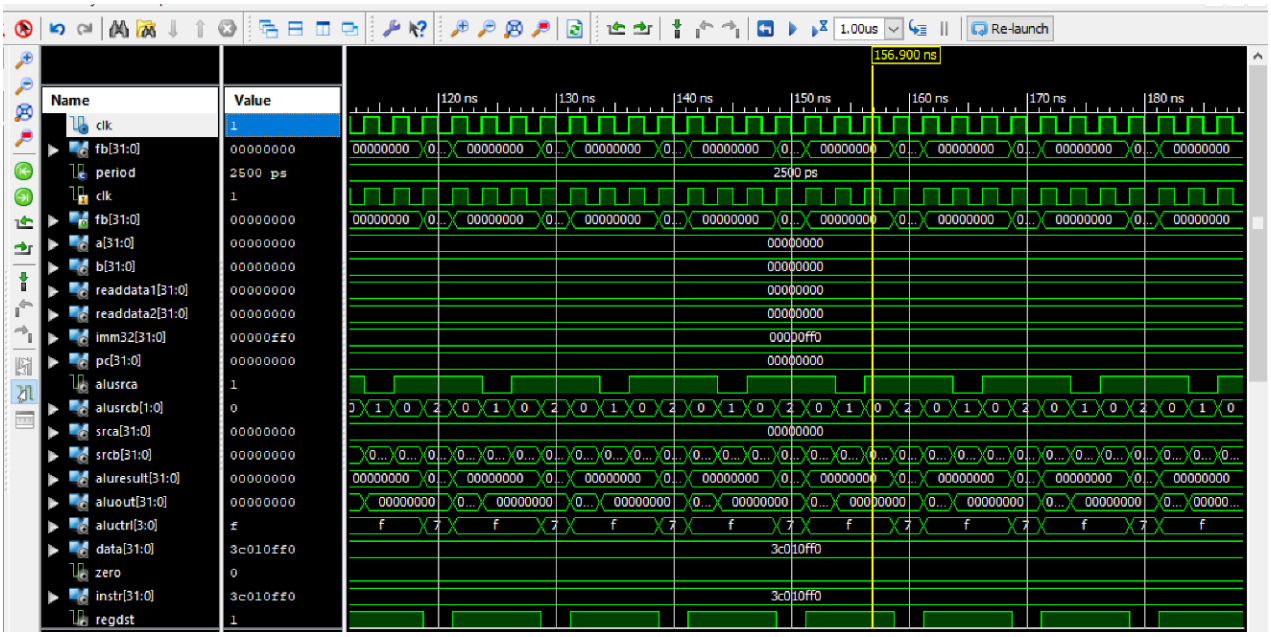
مطابق انتظار، در R_3 مقدار 6 از سیگنال writedata ذخیره شده و مقادیر 4 و 5 که در R_1 و R_2 ذخیره شده اند روی سیگنال readdata1,2 ریخته شده اند.

-

در نهایت هم یک بار عدد 16 بیتی مثبت و بار دیگر عددی منفی را به ماژول SgnExt داده و دیدیم که چگونه به 32 بیت گسترش یافتند.



برای دیدن خروجی نهایی، در testbench مربوط به Top Module به کلاک مقدار داده و مقادیر سیگنال‌های میانی ماژول‌ها را در Isim مشاهده کردیم: (دستور اول که lui بود، کاملا درست اجرا شده.)



-

لینک منابع مورد استفاده:

1. Implementing an FSM in VHDL – AllAboutCircuits
2. MIPS Instruction formats – Kalamazoo College
3. Introduction of Control Unit and its Design – Geeks for Geeks
4. Encoding MIPS Instructions – University of Porto
5. Most MIPS I(TM) opcodes - OpenCore
6. Implement instructions to multicycle datapath – Stack Overflow
7. The mfhi and mflo Instructions - Central Connecticut State University
8. Computer Architecture Worksheet – Alexandria University
9. Multi Cycle CPU - AmrikSadhra

تقسیم کار پروژه:

پیاده سازی کد مربوط به مسیر داده (رجیسترها و مالتی‌پلکسرها، ALU، حافظه و رجیسترفایل)، نوشتن کد Top level و اتصال ماژول‌ها، نوشتن و رفع اشکال تست‌بنچ مربوط به ماژول‌های مسیر داده و تست‌بنچ نهایی، نوشتن توضیحات گزارش مربوط به مسیر داده : زهرا لطیفی

پیاده‌سازی کد مربوط به واحد کنترلی، طراحی و پیاده‌سازی FSM، نوشتن کد Assembly برای تست، نوشتن گزارش مربوط به واحد کنترلی و Top level و سایر توضیحات، تست‌بنچ واحد کنترلی و مالتی‌پلکسر، ضبط ویدیو: مریم مقتدری