دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی برق

پروژه میانترم درس مدارهای منطقی برنامه پذیر

گزارشکار پیاده سازی و شبیه سازی پردازنده MIPS (Multi cycle)

نگارش

مهدیه سادات بنیس

معصومه محمدخانی

استاد درس

دکتر محمدرضا پورفرد

اردیبهشت ۱۴۰۲

چکیده

پردازنده MIPS Multi-cycle یک پردازنده دستورالعمل ثابت (RISC) است که برای پردازش داده‌های ۳۲ بیتی طراحی شده است. در این پردازنده، دستورات به صورت چرخه‌های گوناگون اجرا می‌شوند و هر چرخه شامل چندین مرحله است.
ماژول‌های این پردازنده شامل ماژول کنترل‌کننده، ماژول حافظه، ماژول آدرس‌دهی، ماژول ALU(واحد منطقی حسابی)، ماژول رجیسترها و ماژول‌های مربوط به دستورات مختلف مانند دستورات حرکت داده، دستورات مقایسه‌ای و دستورات فراخوانی هستند.

ماژول کنترل‌کننده مسئول مدیریت چرخه‌ها و کنترل اجرای دستورات است. ماژول حافظه برای ذخیره داده‌ها و دستورات استفاده می‌شود و ماژول آدرس‌دهی مسئول محاسبه آدرس‌های حافظه مورد نیاز است.
ماژول ALU، دستورات حسابی و منطقی را اجرا می‌کند و ماژول ثبات‌ها، مقادیر موقت را در هنگام اجرای چرخه‌های مختلف نگهداری می‌کنند.
دستورات حرکت داده برای انتقال داده‌ها بین ثبات‌ها و حافظه، دستورات مقایسه‌ای برای مقایسه داده‌ها و انجام شرطی، و دستورات فراخوانی برای فراخوانی زیربرنامه‌ها و اجرای آن‌ها استفاده می‌شوند.

به طور کلی، پردازنده MIPS Multi-cycle با استفاده از ماژول‌های مختلف، دستورات مختلف را اجرا می‌کند و به صورت چرخه‌ای و با دقت بالا، داده‌های ۳۲ بیتی را پردازش می‌کند.

در این پروژه قصد داریم این پردازنده را با کد VHDL در محیط ise طراحی و پیاده سازی کنیم. در انتها با نوشتن تست بنچ آن را اجرا کرده و نتایج شبیه سازی و همچنین توضیحات بیشتر را ضبط کردیم.
به علت حجم بالا فایل کدها و ویدیو ها در گوگل درایو بارگذاری و لینک آن در ادامه قرار گرفته شده است.

https://drive.google.com/drive/folders/۱a۳_۳JbpZjyBzwJcjMCc۱gn۹CqCRmEzz۵?usp=share_link

صفحه

فهرست مطالب

فصل اول

مقدمه

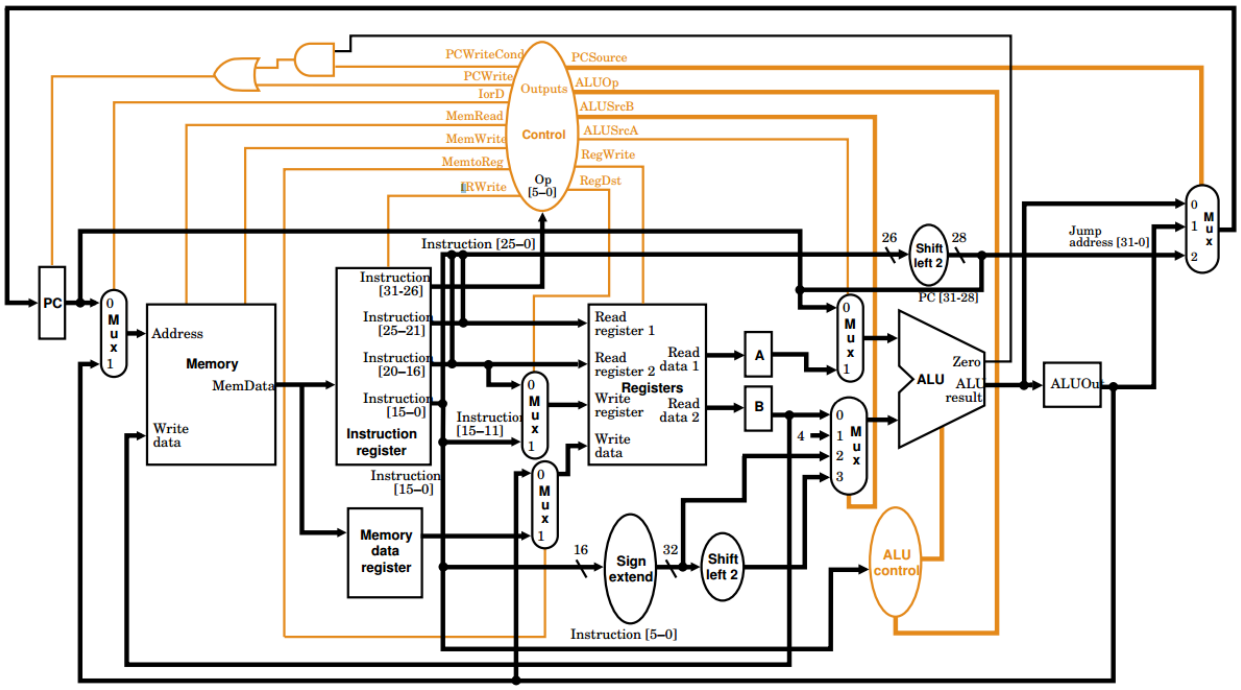
پردازنده Multi-Cycle MIPS یک پردازنده کامپیوتری است که برای اجرای دستورات مختلف در برنامه های کامپیوتری استفاده می شود. این پردازنده بر اساس معماری MIPS[1] طراحی شده است که یک معماری پردازشی RISC[2] است.

در پردازنده Multi-Cycle MIPS، هر دستور به چندین دور (Cycle) تقسیم می شود که هر دور شامل یک مرحله از اجرای دستور است. به عنوان مثال، برای اجرای یک دستور ساده مانند افزایش مقدار یک رجیستر، سه دور زمانی (Cycle) نیاز است. در دور اول، دستور از حافظه خوانده می شود، در دور دوم، مقدار دستوری Opcode و آدرس رجیستر مورد نظر استخراج می شود و در دور سوم، مقدار اولیه رجیستر به علاوه یک می شود و نتیجه در رجیستر مورد نظر ذخیره می شود.
این روش پردازش چند دوره‌ای، در مقایسه با روش پردازش یک دوره‌ای، به دلیل اینکه هر دستور به چندین دور زمانی برای اجرا نیاز دارد، باعث کاهش فرکانس سیستم و افزایش زمان اجرای دستورات می‌شود. اما به دلیل سادگی طراحی و امکان پیاده‌سازی سخت‌افزاری آسان، پردازنده Multi-Cycle MIPS به عنوان یک راه‌حل مناسب برای سیستم‌هایی با محدودیت منابع و امکانات محدود مانند Embedded Systems محسوب می‌شود. همچنین، این روش پردازش در مقایسه با روش پردازش یک دوره‌ای، بهبود قابل توجه‌ای در عملکرد و قابلیت اطمینان سیستم برای اجرای دستورات پیچیده مانند عملیات‌های حسابی و منطقی ارائه می‌دهد.

پردازنده Multi-Cycle MIPS شامل ماژول‌های مختلفی است که هر کدام وظایف مشخصی را در پردازش دستورات انجام می‌دهند. شکل ۱-۱ نمایشگر شمای کلی از قسمت های مختلف این پردازنده است.

۱ Microprocessor without Interlocked Pipelined Stages

۲ Reduced Instruction Set Computer



شکل ۱-۱ شمای کلی از ماژول های پردازنده Multi-Cycle MIPS

در ادامه به طور خلاصه، ماژول‌های مختلف در پردازنده Multi-Cycle MIPS را توضیح می‌دهیم:

۱. Instruction Memory: این ماژول برای خواندن دستورات از حافظه استفاده می‌شود. دستورات در حافظه بر پایه آدرس ذخیره شده‌اند و در هر دوره زمانی، دستور بعدی از حافظه خوانده می‌شود.

۲. Register File: این ماژول برای ذخیره و خواندن مقادیر رجیسترها استفاده می‌شود. رجیسترها می‌توانند به عنوان منابع و مقصد دستورات استفاده شوند و همچنین مقادیر محاسباتی در آن‌ها ذخیره می‌شوند.

۳. ALU[۱]: این ماژول برای انجام عملیات‌های حسابی و منطقی استفاده می‌شود. در پردازش دستورات، مقادیر رجیسترها با استفاده از ALU پردازش می‌شوند و نتیجه در رجیسترها ذخیره می‌شود.

۴. Data Memory: این ماژول برای خواندن و نوشتن داده‌ها به حافظه استفاده می‌شود. داده‌ها در حافظه بر پایه آدرس ذخیره شده‌اند و با استفاده از این ماژول می‌توان آن‌ها را خواند یا در آن‌ها نوشت.

۱ Arithmetic Logic Unit

۵. Control Unit: این ماژول برای کنترل جریان دستورات و سیگنال‌های سخت‌افزاری در پردازنده استفاده می‌شود. این ماژول بر اساس نوع دستورات و وضعیت‌های دیگر در سیستم، سیگنال‌های کنترلی را ایجاد می‌کند و به ماژول‌های دیگر ارسال می‌کند تا پردازش دستورات به درستی انجام شود.

۶. PC[1]: این ماژول برای حفظ آدرس دستور بعدی که باید اجرا شود، استفاده می‌شود. در هر دوره زمانی، آدرس دستور بعدی از PC خوانده می‌شود و PC به آدرس دستور بعدی به روزرسانی می‌شود.

۷. Sign-Extend: این ماژول برای تبدیل مقدار ۱۶ بیتی به ۳۲ بیتی استفاده می‌شود. این ماژول به عنوان ورودی یک مقدار ۱۶ بیتی دریافت می‌کند و به عنوان خروجی یک مقدار ۳۲ بیتی با افزودن صفرهای لازم به سمت چپ آن را تولید می‌کند.

در کل، پردازنده Multi-Cycle MIPS شامل ماژول‌های مختلفی است که هر کدام وظایف و عملکردهای مختلفی در پردازش دستورات انجام می‌دهند. با ترکیب این ماژول‌ها، پردازنده Multi-Cycle MIPS به عنوان یک سیستم کامل پردازشی می‌تواند دستورات پیچیده را با سرعت و دقت بالا اجرا کرده و عملکرد مناسبی را به نمایش بگذارد.

در فصل های بعد نحوه پیاده سازی این ماژول ها با کد VHDL را توضیح میدهیم.

۱ Program Counter

فصل دوم
پیاده سازی Instruction Memory و Memory data register

در این بخش به نحوه پیاده سازی و توضیحات تکمیلی در رابطه با ماژول های Instruction Memory و Memory data register میپردازیم.

۲-۱ Instruction Memory:

ماژول "Instruction Memory" یک ماژول حافظه است که برای ذخیره و بازیابی دستورات برنامه‌های MIPS استفاده می‌شود. این ماژول به عنوان یکی از اجزای اصلی پردازنده MIPS، دستورات برنامه را از حافظه دریافت می‌کند و سپس آن‌ها را به بخش کنترل پردازنده می‌فرستد تا برای اجرا آماده شوند.

در طراحی این ماژول، می‌توان از یک آرایه حافظه استفاده کرد که دستورات برنامه در آن ذخیره شده‌اند. برای مثال، در یک سیستم با حافظه ۴ کیلوبایتی، این آرایه می‌تواند شامل ۱۰۲۴ خانه باشد که هر خانه یک دستور ۴ بایتی (۳۲ بیتی) را نشان می‌دهد.
در فرآیند اجرای برنامه، با توجه به شماره دستور (instruction address) که توسط بخش کنترل پردازنده تولید می‌شود، آدرس حافظه مورد نظر انتخاب می‌شود و دستور موجود در آن آدرس از حافظه استخراج می‌شود و به بخش کنترل پردازنده ارسال می‌شود.

ماژول Instruction Memory می‌تواند به صورت سریال یا پارالل پیاده‌سازی شود. در حالت سریال، دستورات برنامه به صورت یکی پس از دیگری از حافظه استخراج می‌شوند و در حالت پارالل، چند دستور همزمان از حافظه استخراج می‌شوند. در هر دو حالت، حافظه دستورات باید بتواند دستورات را به صورت سریع و با دقت بالا بازیابی کند تا پردازنده بتواند برنامه را با سرعت بالا اجرا کند.

برای پیاده سازی این ماژول با کد VHDL یک ماژول با نام Memory_inst تعریف کردیم که شامل ورودی و خروجی های زیر است:
ورودی:

- memory_read
- memory_write
- address
- write_data
- clk
- rst

خروجی:

- memory_data

حافظه با استفاده از آرایه mem به صورت یک حافظه ۶۴ بایتی پیاده‌سازی شده است. هر خانه از حافظه به عنوان یک رشته باینری به طول ۸ بیتی نشان داده شده است.

برای نوشتن در حافظه، در فرآیندی که با کلاک سینکرونیزه شده است، اگر memory_write برابر با ۱ باشد و کلاک رو به بالا باشد، داده‌های write_data در آدرس مشخص شده توسط address در حافظه نوشته می‌شوند. برای خواندن از حافظه، اگر memory_read برابر با ۱ باشد، داده‌های موجود در آدرس مشخص شده توسط address در حافظه خوانده می‌شوند و در memory_data قرار داده می‌شوند. برای این منظور، داده‌های ۴ بایتی (۳۲ بیتی) موجود در ۴ خانه پشت سر هم در حافظه، به صورت یک رشته ۳۲ بیتی در memory_data قرار داده می‌شوند.

در بخشی از کد که با کامنت auto generated مشخص شده است، داده‌های باینری مربوط به چند دستور MIPS ذخیره شده‌اند که می‌توانند در شبیه‌سازی پردازنده MIPS به عنوان داده‌های ورودی مورد استفاده قرار بگیرند.

قسمت هایی از کد در صفحات بعد آورده شده است.

```
entity Memory is
    Port ( memory_read : in  STD_LOGIC;
           memory_write : in  STD_LOGIC;
           address : in  STD_LOGIC_VECTOR (31 downto 0);
           write_data : in  STD_LOGIC_VECTOR (31 downto 0);
           clk : in  STD_LOGIC;
           rst: in  STD_LOGIC;
           memory_data : out  STD_LOGIC_VECTOR (31 downto 0));
end Memory;

architecture Behavioral of Memory is

type A is  array ( 0 to 63) of STD_LOGIC_VECTOR(7 downto 0);
signal mem: A := (
-- auto generated
-- lw $R0,47($R20)
0 => "10001110",
1 => "10000000",
2 => "00000000",
3 => "00101111",

-- addi $R1,$R3,50
4 => "00100000",
5 => "01100001",
6 => "00000000",
7 => "00110010",

-- addi $R2,$R3,48
8 => "00100000",
9 => "01100010",
10 => "00000000",
11 => "00110000",

-- add $R2,$R2,$R0
12 => "00000000",
13 => "01000000",
14 => "00010000",
15 => "00100000",

-- beq $R1,$R2,1
16 => "00010000",
17 => "00100010",
18 => "00000000",
19 => "00000001",

-- j 3
20 => "00001000",
21 => "00000000",
22 => "00000000",
23 => "00000011",

-- add $R0,$R1,$R2
24 => "00000000",
```

```
begin

   process(clk, rst)
   begin
      ---if rst = '0' then
      ---   mem <= (others => (others => '0'));
      ---end if;
      if rising_edge(clk) and memory_write = '1' then
         mem(to_integer(unsigned(address))) <= write_data(31 downto 24);
         mem(to_integer(unsigned(address) + 1)) <= write_data(23 downto 16);
         mem(to_integer(unsigned(address) + 2)) <= write_data(15 downto 8);
         mem(to_integer(unsigned(address) + 3)) <= write_data(7 downto 0);
      end if;
   end process;

   memory_data <= mem(to_integer(unsigned(address))) &
                  mem(to_integer(unsigned(address) + 1)) &
                  mem(to_integer(unsigned(address) + 2)) &
                  mem(to_integer(unsigned(address) + 3)) when memory_read = '1';
end Behavioral;
```

۲-۲ Memory data register:

Memory Data Register به عنوان یکی از اجزای اصلی یک سیستم پردازشی، برای ذخیره داده‌هایی استفاده می‌شود که باید در مراحل بعدی پردازش توسط پردازنده استفاده شوند. به‌عنوان مثال، وقتی که یک دستور برنامه توسط ماژول Instruction Memory دریافت شده، داده‌هایی که در این دستور استفاده شده‌اند در رجیستر Memory Data Register ذخیره می‌شوند تا در مراحل بعدی پردازش توسط پردازنده استفاده شوند.

برای پیاده‌سازی این ماژول که یک رجیستر ۳۲ بیتی است و در هر لبه مثبت کلاک، داده‌های ورودی را دریافت کرده و در خروجی خود قرار می‌دهد، ورودی ها و خروجی های زیذ را تعریف کرده‌ایم:

ورودی:

- clk: کلاک سیستم
- rst: سیگنال ریست که در صورت فعال شدن، مقدار رجیستر را صفر می‌کند.
- mem_input: داده‌هایی که از حافظه خوانده شده‌اند.

خروجی:

- mem_output: داده‌ای که در رجیستر ذخیره شده است.

در طراحی این ماژول، از یک آرایه یک بعدی با طول ۱ به نام MemDataReg استفاده شده است. این آرایه، یک رجیستر ۳۲ بیتی را نشان می‌دهد که در هر لبه مثبت کلاک، داده‌های ورودی را دریافت کرده و در خود ذخیره می‌کند.

در فرآیند اجرای کد، اگر سیگنال rst فعال شده باشد، مقدار رجیستر صفر می‌شود. در غیر این صورت، اگر کلاک رو به بالا باشد، داده‌های ورودی در رجیستر ذخیره می‌شوند و در خروجی قرار داده می‌شوند.

این ماژول به صورت سریال پیاده‌سازی شده است، به این معنی که تنها یک رجیستر وجود دارد و در هر لبه مثبت کلاک، داده‌های ورودی در آن ذخیره می‌شوند. اما به راحتی می‌توان این ماژول را به صورت پارالل پیاده‌سازی کرد، به این معنی که چندین رجیستر وجود داشته باشد و در هر لبه مثبت کلاک، داده‌های ورودی به همه‌ی این رجیسترها همزمان داده شوند و در آنها ذخیره شوند.

کد های این بخش در ادامه قابل مشاهده‌اند.

```
entity MemoryDataRegister is
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           mem_input : in  STD_LOGIC_VECTOR (31 downto 0);
           mem_output : out  STD_LOGIC_VECTOR (31 downto 0));
end MemoryDataRegister;

architecture Behavioral of MemoryDataRegister is

type mem_data_type is array (0 downto 0) of std_logic_vector(31 downto 0);
signal MemDataReg: mem_data_type := ((others => (others => '0')));

begin

   process(clk)
   begin
      if rst = '0' then
         MemDataReg(0) <= (others => '0');
      else if rising_edge(clk) then
         MemDataReg(0) <= mem_input;
      end if;
      end if;
   end process;
   mem_output <= MemDataReg(0);

end Behavioral;
```

فصل سوم

پیاده سازی Instruction Register و Register File

در این بخش به نحوه پیاده سازی و توضیحات حول ماژول‌های Instruction Register و Register File میپردازیم.

۳-۱ Instruction Register:

Instruction Register برای ذخیره دستوراتی استفاده می‌شود که از حافظه دریافت شده‌اند و باید توسط پردازنده اجرا شوند. به عنوان مثال، وقتی که یک دستور توسط ماژول Instruction Memory دریافت شده، آن دستور در رجیستر Instruction Register ذخیره می‌شود تا در مراحل بعدی پردازش توسط پردازنده، اجرا شود.

برای پیاده‌سازی ماژول Instruction Register یک رجیستر ۳۲ بیتی تعریف شده که در هر لبه مثبت کلاک، دستورات ورودی را دریافت کرده و در خروجی خود قرار می‌دهد.

ورودی:

- clk: کلاک سیستم
- rst: سیگنال ریست که در صورت فعال شدن، مقدار رجیستر را صفر می‌کند.
- IRWrite: سیگنالی که نشان می‌دهد آیا امکان نوشتن در رجیستر وجود دارد یا نه.
- input_instruction: دستوراتی که از حافظه خوانده شده‌اند.

خروجی:

- output_instruction: دستوری که در رجیستر ذخیره شده است.

در طراحی این ماژول، از یک آرایه یک بعدی با طول ۱ به نام instr_reg استفاده شده است. این آرایه، یک رجیستر ۳۲ بیتی را نشان می‌دهد که در هر لبه مثبت کلاک، دستورات ورودی را دریافت کرده و در خود ذخیره می‌کند.

در فرآیند اجرای کد، اگر سیگنال rst فعال شده باشد، مقدار رجیستر صفر می‌شود. در غیر این صورت، اگر کلاک رو به بالا باشد و سیگنال IRWrite فعال باشد، دستورات ورودی در رجیستر ذخیره می‌شوند و در خروجی قرار داده می‌شوند.

کد های استفاده شده برای پیاده سازی این بخش در ادامه قرار گرفته‌اند.

```
entity InstructionRegister is
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           IRWrite : in  STD_LOGIC;
           input_instruction : in  STD_LOGIC_VECTOR (31 downto 0);
           output_instruction : out  STD_LOGIC_VECTOR (31 downto 0));
end InstructionRegister;

architecture Behavioral of InstructionRegister is
type instr_reg_type is array (0 to 0) of std_logic_vector(31 downto 0);
signal instr_reg : instr_reg_type := ((others => (others => '0')));

begin

   process(clk)
   begin
      if rst = '0' then
         instr_reg(0) <= (others => '0');
      else if rising_edge(clk) and IRWrite = '1' then
         instr_reg(0) <= input_instruction;
      end if;
      end if;
   end process;
   output_instruction <= instr_reg(0);
end Behavioral;
```

Register File ۲-۳

ماژول Register File برای ذخیره داده‌های مختلف استفاده می‌شود. این داده‌ها ممکن است شامل مقادیر محاسباتی، اطلاعات ورودی، نتایج محاسباتی و غیره باشند. با استفاده از این رجیسترها، می‌توان داده‌ها را ذخیره کرد و در مراحل بعدی پردازش، از آنها استفاده کرد.

برای پیاده سازی یک ماژول Registers را طراحی کردیم که برای ذخیره داده‌های پردازشگر استفاده می‌شود. این ماژول شامل ۳۲ رجیستر ۳۲ بیتی است که در هر لبه مثبت کلاک، داده‌های ورودی را دریافت کرده و در خروجی خود قرار می‌دهد.

ورودی:

- clk: کلاک سیستم
- rst: سیگنال ریست که در صورت فعال شدن، مقدار تمامی رجیسترها را صفر می‌کند.
- read_register۱: شماره رجیستری که باید از آن خوانده شود.
- read_register۲: شماره رجیستری دوم که باید از آن خوانده شود.
- write_register: شماره رجیستری که باید در آن نوشته شود.
- write_data: داده‌ای که باید در رجیستر نوشته شود.
- RegWrite: سیگنالی که نشان می‌دهد آیا امکان نوشتن در رجیستر وجود دارد یا نه.

خروجی:

- read_data۱: داده‌ای که از رجیستر با شماره read_register۱ خوانده شده است.
- read_data۲: داده‌ای که از رجیستر با شماره read_register۲ خوانده شده است.

در طراحی این ماژول، از یک آرایه یک بعدی با طول ۳۲ به نام reg استفاده شده است. این آرایه، ۳۲ رجیستر ۳۲ بیتی را نشان می‌دهد که در هر لبه مثبت کلاک، داده‌های ورودی را دریافت کرده و در خود ذخیره می‌کند.

در فرآیند اجرای کد، اگر سیگنال rst فعال شده باشد، تمامی رجیسترها صفر می‌شوند. در غیر این صورت، اگر کلاک رو به بالا باشد و سیگنال RegWrite فعال باشد، داده‌های ورودی در رجیستر مربوطه ذخیره می‌شوند.

برای خواندن داده‌ها از رجیسترها، شماره رجیسترهایی که باید از آنها خوانده شود را با استفاده از ورودی‌های read_register۱ و read_register۲ به ماژول داده و داده‌های موجود در آنها در خروجی‌های read_data۱ و read_data۲ قرار داده می‌شوند.

کد نوشته شده برای این قسمت را در ادامه قرار داده‌ایم:

```
entity Registers is
    Port ( clk : in  STD_LOGIC;
           rst : in  STD_LOGIC;
           read_register1 : in  STD_LOGIC_VECTOR (4 downto 0);
           read_register2 : in  STD_LOGIC_VECTOR (4 downto 0);
           write_register : in  STD_LOGIC_VECTOR (4 downto 0);
           write_data : in  STD_LOGIC_VECTOR (31 downto 0);
           RegWrite : in  STD_LOGIC;
           read_data1 : out  STD_LOGIC_VECTOR (31 downto 0);
           read_data2 : out  STD_LOGIC_VECTOR (31 downto 0));
end Registers;

architecture Behavioral of Registers is

type reg_type is array (0 to 31) of std_logic_vector(31 downto 0);
signal reg : reg_type := (others => (others => '0'));

begin

   process(clk)
   begin
      if rst = '0' then
         reg(to_integer(unsigned(write_register))) <= (others => '0');
      else if rising_edge(clk) and RegWrite = '1' then
         reg(to_integer(unsigned(write_register))) <= write_data;
      end if;
      end if;
   end process;
```

فصل چهارم

پیاده سازی ALU و ALU control

دستوراتی که دریافت میشوند ابتدا نیاز به کدگشایی دارند تا مشخص شود چه عملیاتی باید روی آنها انجام شود. در این بخش ابتدا به انواع دستورات در پردازنده MIPS میپردازیم.

۴-۱ انواع دستورات در پردازنده MIPS

مجموعه دستورات MIPS یک ساختار کامپیوتر با معماری RISC است که در بسیاری از پردازنده‌های مدرن استفاده می‌شود. دستورات MIPS شامل سه نوع دستور با طول ۳۲ بیتی هستند.

۱. دستورات نوع R: این دستورات برای انجام عملیات‌های حسابی و منطقی استفاده می‌شوند. فرمت این دستورات به صورت زیر است:

opcode[31:26]	Rs[25:21]	Rt[20:16]	Rd[15:11]	Shamt[10:6]	Function[5:0]

- Opcode: کد عملیات را مشخص می‌کند.
- Rs: ثبت منبع ۱.
- Rt: ثبت منبع ۲.
- Rd: ثبت مقصد.
- Shamt: مقدار شیفت.
- Function: کد تابع را مشخص می‌کند.

۲. دستورات نوع I: این دستورات برای انجام عملیات‌های فوری استفاده می‌شوند، مانند بارگذاری ثابت در ثبت‌ها یا شاخه‌گیری. فرمت این دستورات به صورت زیر است:

opcode[31:26]	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

- Opcode: کد عملیات را مشخص می‌کند.
- Rs: ثبت منبع.
- Rt: ثبت مقصد.
- Address/constant: مقدار فوری.

۳. دستورات نوع J: این دستورات برای پرش به یک آدرس حافظه خاص استفاده می‌شوند. فرمت این دستورات به صورت زیر است:

opcode[31:26]	Address[25:0]

- Opcode: کد عملیات را مشخص می‌کند.
- Address: آدرس ۲۶ بیتی برای پرش.

بعضی از دستورات رایج در مجموعه دستورات MIPS عبارتند از:

J-type:

۱. jump – j instruction

به یک آدرس حافظه خاص پرش می‌کند.

000010	Address[25:0]

Operation: PC ← PC[31:28] || Inst[25:0] || 00

۲. jump and link – jal instruction

به یک آدرس حافظه خاص پرش می‌کند و آدرس بازگشتی را در یک رجیستر ذخیره می‌کند.

000011	Address[25:0]

Operation: $ra ← PC + 1
PC ← PC[31:28] || Inst[25:0] || 00

R-type:

۱. Jump and link register – jalr instruction

به یک آدرسی از حافظه که در رجیستری مشخص است پرش می‌کند و آدرس بازگشتی را در یک رجیستر مشخص ذخیره می‌کند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۰۰۱۰۰۱

Operation: PC ← Rs
Rd ← PC + 1

۲. Jump register – jr instruction

به آدرسی که در یک ثبت ذخیره شده است پرش می‌کند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۰۰۱۰۰۰

Operation: PC ← Rs

۳. Addition – add instruction

دو رجیستر را با هم جمع می‌کند و نتیجه را در یک رجیستر دیگر ذخیره می‌کند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۰۰۰

Operation: Rd ← Rs + Rt
PC ← PC + 1

۴. Unsigned addition – addu instruction

مقدار دو رجیستر را بدون علامت با هم جمع می‌کند و نتیجه را در یک رجیستر دیگر ذخیره می‌کند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۰۰۱

Operation: Rd ← Rs + Rt
PC ← PC + 1

۵. Subtraction – sub instruction

مقدار یک رجیستر را از یک رجیستر دیگر کم می‌کند و نتیجه را در رجیستر سوم ذخیره می‌کند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۰۱۰

Operation: Rd ← Rs - Rt
PC ← PC + 1

۶. Unsigned subtraction – subu instruction

مقدار یک رجیستر را از یک رجیستر دیگر بدون علامت کم می‌کند و نتیجه را در رجیستر سوم ذخیره می‌کند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۰۱۱

Operation: Rd ← Rs - Rt
PC ← PC + 1

۷. Logical AND – AND instruction

عملیات منطقی AND را بین دو رجیستر اول و دوم انجام و نتیجه را در رجیستر سوم ذخیره میکند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۱۰۰

Operation: Rd ← Rs AND Rt
PC ← PC + 1

۸. Logical OR – OR instruction

عملیات منطقی OR را بین دو رجیستر اول و دوم انجام و نتیجه را در رجیستر سوم ذخیره میکند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۱۰۱

Operation: Rd ← Rs OR Rt
PC ← PC + 1

۹. Logical XOR – XOR instruction

عملیات منطقی XOR را بین دو رجیستر اول و دوم انجام و نتیجه را در رجیستر سوم ذخیره میکند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۱۱۰

Operation: Rd ← Rs XOR Rt
PC ← PC + 1

۱۰. Logical NOR – NOR instruction

عملیات منطقی NOR را بین دو رجیستر اول و دوم انجام و نتیجه را در رجیستر سوم ذخیره میکند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۰۱۱۱

Operation: Rd ← Rs NOR Rt
PC ← PC + 1

۱۱. Set if less than unsigned – sltu instruction

رجیستر اول را با رجیستر دوم بدون علامت مقایسه میکند، اگر کوچکتر بود ۱ و در غیر این صورت ۰ را در رجیستر مقصد ذخیره میکند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۱۰۰۱

Operation: if Rs < Rt then Rd ← 1
Else Rd ← 0
PC ← PC + 1

۱۲. Set if less than: slt instruction

رجیستر اول را با رجیستر دوم با علامت مقایسه میکند، اگر کوچکتر بود ۱ و در غیر این صورت ۰ را در رجیستر مقصد ذخیره میکند.

۰۰۰۰۰۰	Rs[25:21]	Rt[20:16]	Rd[15:11]	۰۰۰۰۰	۱۰۱۰۱۰

Operation: if Rs < Rt then Rd ← 1
Else Rd ← 0
PC ← PC + 1

I-type:

۱. Branch on equal: beq instruction

در صورت مساوی بودن دو رجیستر، به آدرس حافظه دیگری پرش می‌کند.

۰۰۰۱۰۰	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: if Rs = Rt then PC ← PC + 1 + ((sign extended I[15:0]) || 00)
else PC ← PC + 1

۲. Branch on not equal: bne instruction

در صورت مساوی نبودن دو رجیستر، به آدرس حافظه دیگری پرش می‌کند.

۰۰۰۱۰۱	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: if Rs ≠ Rt then PC ← PC + 1 + ((sign extended I[15:0]) || 00)
else PC ← PC + 1

۳. Immediate addition: addi instruction

مقدار رجیستر اول را به مقدار ثابت اضافه کرده و در رجیستر دوم ذخیره میکند.

۰۰۱۰۰۰	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: Rt ← Rs + (sign extended I[15:0])
PC ← PC +1

۴. Immediate addition unsigned: addiu instruction

مقدار رجیستر اول را با مقدار ثابت بدون علامت جمع کرده و در رجیستر دوم ذخیره میکند.

۰۰۱۰۰۱	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: Rt ← Rs + (sign extended I[15:0])
PC ← PC +1

۵. Immediate set-if-less-than unsigned: sltiu instruction

مقدار رجیستر اول با مقدار ثابت بدون علامت مقایسه میکند و اگر کمتر بود ۱ و در غیر این صورت ۰ را در رجیستر دوم ذخیره میکند.

۰۰۱۰۱۱	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: if Rs < (sign extended I[15:0]) then Rt ← 1
else Rt ← 0
PC ← PC + 1

۶. Immediate set-if-less-than: slti instruction

مقدار رجیستر اول با مقدار ثابت با علامت مقایسه میکند و اگر کمتر بود ۱ و در غیر این صورت ۰ را در رجیستر دوم ذخیره میکند.

۰۰۱۰۱۰	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: if Rs < (sign extended I[15:0]) then Rt ← 1
else Rt ← 0
PC ← PC + 1

۷. Immediate logic AND: andi instruction

عملیات منطقی AND را بین رجیستر اول و مقدار ثابت انجام میدهد و نتیجه آن را در رجیستر دوم ذخیره میکند.

۰۰۱۱۰۰	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: Rt ← Rs AND (sign extended I[15:0])
PC ← PC + 1

۸. Immediate logic OR: ori instruction

عملیات منطقی OR را بین رجیستر اول و مقدار ثابت انجام میدهد و نتیجه آن را در رجیستر دوم ذخیره میکند.

۰۰۱۱۰۱	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: Rt ← Rs OR (sign extended I[15:0])
PC ← PC + 1

۹. Immediate logic XOR: xori instruction

عملیات منطقی XOR را بین رجیستر اول و مقدار ثابت انجام میدهد و نتیجه آن را در رجیستر دوم ذخیره میکند.

۰۰۱۱۱۰	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: Rt ← Rs XOR (sign extended I[15:0])
PC ← PC + 1

۱۰. Load Word: lw instruction

مقدار ذخیره شده در آدرس را با آفست رجیستر اول، در رجیستر دوم بارگذاری میکند.

۱۰۰۰۱۱	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: Rt ← M[Rs + (sign extended I[15:0])]
PC ← PC + 1

۱۱. Store Word: sw instruction

مقدار رجیستر دوم را در آدرس به اضافه آفست رجیستر اول، ذخیره میکند.

۱۰۱۰۱۱	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

Operation: M[Rs + (sign extended I[15:0])] ← Rt
PC ← PC + 1

۱۲. Load upper immediate: LUI instruction

مقدار ثابت را ۱۶ بیت به چپ شیفت داده و ۱۶ بیت پایین را صفر میکند و مقدار نهایی را در رجیستر RT ذخیره میکند.

۰۰۱۱۱۱	Rs[25:21]	Rt[20:16]	Address/constant [15:0]

در بخش های بعد این دستورات را پیاده سازی میکنیم.

۲-۴ پیاده سازی ALU control

قصد داریم یک واحد کنترل ALU را پیاده‌سازی کنیم.

ورودی:

- Instruction: یک بردار ۶ بیتی است که شامل کد عملیات مورد نظر برای ALU است.
- ALUOp: یک بردار ۳ بیتی است که شامل کد مورد نظر برای کنترل ALU است.

خروجی:

- output

دو متغیر signal با نام operation و temp تعریف کردیم. temp برای ذخیره کد عملیات مورد نظر برای ALU در حالت‌های مختلف ALUOp و operation برای ذخیره کد دقیق عملیات ALU است که بر اساس instruction تعیین می‌شود.

کد Behavioral، شامل یک فرایند با استفاده از دستور select است که با استفاده از ALUOp، کد عملیات مورد نیاز برای ALU را در موارد مختلف تعیین می‌کند. به عنوان مثال، در صورتی که ALUOp برابر با "۰۰۰" باشد، کد عملیات برای addi(جمع با یک مقدار فوری) تعیین می‌شود و در صورتی که برابر با "۱۰۰" باشد، کد عملیات برای andi تعیین می‌شود.

در قسمت دیگری از این فرایند، به کمک دستور select دیگری، کد دقیق عملیات ALU برای هر instruction ممکن تعیین می‌شود. به عنوان مثال، در صورتی که instruction برابر با "۱۰۰۰۰۰" باشد، کد عملیات دقیق برای جمع (add) تعیین می‌شود و در صورتی که instruction برابر با "۱۰۰۰۱۰" باشد، کد عملیات برای تفریق (sub) تعیین می‌شود.

در نهایت، مقدار temp که کد عملیات برای ALU را در بر داشت، به عنوان خروجی output ارسال می‌شود.

کد مورد استفاده در ادامه قرار داده شده است:

```
entity ALUControl is
    Port ( instruction  : in  STD_LOGIC_VECTOR (5 downto 0);
           ALUOp        : in  STD_LOGIC_VECTOR (2 downto 0);
           output       : out STD_LOGIC_VECTOR (3 downto 0));
end ALUControl;

architecture Behavioral of ALUControl is
signal temp, operation, operation_i : std_logic_vector(3 downto 0) := "1111";

begin
     with ALUOp select
    temp <= "0000"      when "000",    -- addi for LW
            "0101"      when "001",    -- xori / Branch
            operation   when "010",    -- R-type
            "0110"      when "011",    -- slti
            "0010"      when "100",    -- andi
            "0011"      when "101",    -- ori
            "0111"      when "110",    -- lui
            "1111"      when others;   -- in other cases

  with instruction select
    operation <= "0000" when "100000", -- add
                 "0001" when "100010", -- sub
                 "0010" when "100100", -- AND
                 "0011" when "100101", -- OR
                 "0100" when "100111", -- NOR
                 "0101" when "100110", -- XOR

                 "0110" when "101010", -- set on less than
                 "1111" when others;
  output <= temp;
end Behavioral;
```

۳-۴ پیاده سازی ALU:

برای این ماژول یک entity به نام ALU را تعریف کردیم که ورودی ها و خروجی های زیر را دارد.

ورودی:

- a و b: بیت‌های عملیات را برای ALU شامل می‌شوند.
- ALU_control: یک بردار ۴ بیتی است که کنترل کننده عملیات ALU را تعیین می‌کند.

خروجی:

- Output: خروجی نهایی ماشین حساب ALU است.
- Zero: آیا خروجی ماشین حساب برابر با صفر است یا نه.

در این کد، با استفاده از یک فرایند با استفاده از دستور select، با توجه به کد عملیات مورد نظر، یک مقدار برای temp تعیین می‌شود که حاوی خروجی نهایی ALU است. به عنوان مثال، در صورتی که کد عملیات برابر با "۰۰۰۰" باشد، مقدارهای a و b باهم جمع شده و در temp ذخیره می‌شوند. در صورتی که کد عملیات برابر با "۰۰۰۱" باشد، مقدار b از a کم می‌شود و در temp ذخیره می‌شود. در موارد دیگر نیز، عملیات مناسب انجام می‌شود و مقدار نهایی در temp ذخیره می‌شود.

در این کد، دو متغیر signal به نام temp و zero تعریف شده‌اند. temp برای ذخیره خروجی نهایی ALU استفاده می‌شود و zero برای نشان دادن اینکه خروجی ماشین حساب برابر با صفر است یانه.

در ادامه، با استفاده از یک فرایند دیگر، مقدار zero بر اساس مقدار temp تعیین می‌شود. در صورتی که temp برابر با صفر باشد، مقدار zero برابر با ۱ قرار داده می‌شود و در غیر این صورت، مقدار zero برابر با ۰ قرار داده می‌شود.

در نهایت، مقدار temp به عنوان خروجی output ارسال می‌شود که در آن، خروجی نهایی ماشین حساب ALU ذخیره شده است.

کد های نوشته شده برای این قسمت در ادامه قابل مشاهده است.

```
entity ALU is
    Port ( a            : in  STD_LOGIC_VECTOR (31 downto 0);
           b            : in  STD_LOGIC_VECTOR (31 downto 0);
           ALU_control  : in  STD_LOGIC_VECTOR (3 downto 0);
           output       : out  STD_LOGIC_VECTOR (31 downto 0);
           zero         : out  STD_LOGIC);
end ALU;

architecture Behavioral of ALU is
 signal temp : std_logic_vector(31 downto 0);

begin

  temp <=

    -- add
    std_logic_vector(unsigned(a) + unsigned(b)) when ALU_control = "0000" else
    -- sub
    std_logic_vector(unsigned(a) - unsigned(b)) when ALU_control = "0001" else
    -- AND
    a AND  b when ALU_control = "0010" else
    -- OR
    a OR   b when ALU_control = "0011" else
    -- NOR
    a NOR  b when ALU_control = "0100" else
    -- XOR
    a XOR b when ALU_control = "0101" else
    -- set on less than
    "00000000000000000000000000000001" when ALU_control = "0110" and (a < b) else
    "00000000000000000000000000000000" when ALU_control = "0110" and (a >= b) else
    -- lui
    (a(15 downto 0) & "0000000000000000") when ALU_control = "0111" else
    -- in other cases
    (others => '0');

  zero <= '1' when temp <= "00000000000000000000000000000000" else '0';
  output <= temp;

end Behavioral;
```

فصل پنجم

پیاده سازی Control Unit

در این فصل نحوه پیاده سازی واحد کنترل را توضیح میدهیم.

با توجه به multi cycle بودن پردازنده، میدانیم که هر دستور در تعداد مشخصی دوره انجام میشوند. برای کنترل دوره‌های مختلف و عملیاتی که باید در آنها انجام شود stateهای مختلفی را تعریف میکنیم که در هر کدام از آنها واحد کنترل خروجی‌های مشخص خود را برای کنترل ماژول‌ها تولید میکند. شمای کلی از stateهای تعریف شده در شکل ۵-۱ نشان داده شده که البته برای اضافه کردن بعضی دستورات، stateهای جدیدی تعریف کردیم.

ورودی:

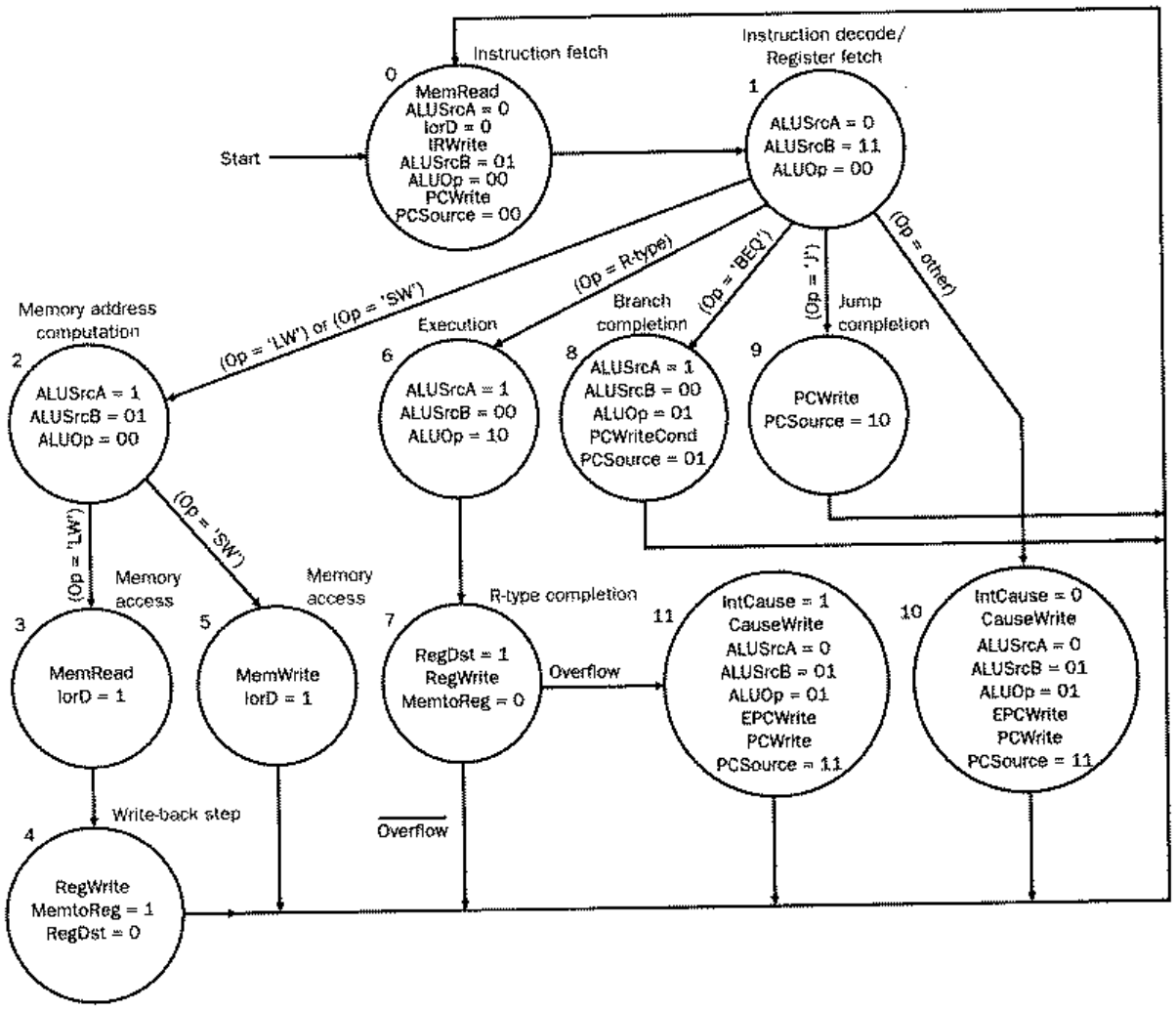
- Clk: کلاک سیستم.
- Rst: سیگنال ریست که در صورت فعال شدن، تمام سیگنال های کنترلی صفر می‌شوند.
- Op: کد opcode
- Func: کد عملکرد دستور در حال اجرا

خروجی:

- سیگنال های کنترلی

در بخش architecture، رفتار واحد کنترل را تعریف کردیم. یک ماشین حالت(State Machine) برای تعیین حالت کنونی واحد کنترل بر اساس دستور فعلی در حال اجرا استفاده می‌شود و سیگنال‌های کنترلی متناظر با هر حالت تولید می‌شوند. ماشین حالت بر اساس کد اپکد و کد عملکرد دستور فعلی و حالت کنونی ماشین، به حالت بعدی منتقل می‌شود.

در بخش process، منطق تغییر حالت بر اساس سیگنال‌های ورودی و تولید سیگنال‌های کنترلی بر اساس حالت کنونی تعریف شده است. با استفاده از دستور with select، حالت کنونی به مجموعه‌ای از سیگنال‌های کنترلی نگاشت می‌شود که به پورت‌های خروجی اختصاص داده می‌شوند.



شکل ۵-۱ شمای کلی از state های مختلف control unit

هر کدام از state ها حالت خاصی را مشخص میکنند که به شکل زیر میتوان عملکرد آنها را تعریف کرد:

۱. Start: حالت شروع برنامه است که در آن واحد کنترل به حالت InstructionFetch منتقل می‌شود تا دستور اول برنامه را دریافت کند. در واقع این حالت موقعی اتفاق میافتد که rst در ورودی مقدار ۱ داشته باشد. برای شروع حتما باید ریست را فعال کنیم زیرا حالت قبلی مولفه ها مشخص نیست.

۲. InstructionFetch: در این حالت، واحد کنترل دستور جدید را از حافظه تعیین می‌کند و به حالت Decode منتقل می‌شود.

۳. Decode: در این حالت، واحد کنترل با استفاده از کد اپکد و کد عملکرد دستور در حال اجرا، سیگنال‌های کنترلی برای اجزای مختلف پردازنده تولید می‌کند و به حالت Execute منتقل می‌شود.

۴. Execute: در این حالت، دستور در حال اجرا است و واحد کنترل برای تولید سیگنال‌های کنترلی برای پردازش دستور، به حالت IncrementPC منتقل می‌شود.

۵. IncrementPC: در این حالت، واحد کنترل شمارنده برنامه را افزایش می‌دهد و به حالت InstructionFetch منتقل می‌شود تا دستور جدید دریافت شود.

۶. Branch: در این حالت، واحد کنترل برای اجرای دستور شرطی، به حالت‌های مختلفی منتقل می‌شود که بسته به نتیجه شرط، به حالت‌های مختلفی از برنامامه منتقل می‌شود.

۷. Jump: در این حالت، واحد کنترل برای اجرای دستور جامپ به حالت Jump۱ منتقل می‌شود.

۸. Jump۱: در این حالت، واحد کنترل شمارنده برنامه را به آدرس مورد نظر تنظیم می‌کند و به حالت InstructionFetch منتقل می‌شود تا دستور جدید دریافت شود.

۹. Load: در این حالت، واحد کنترل برای خواندن داده از حافظه به رجیستر مورد نظر، به حالت MemoryRead منتقل می‌شود.

۱۰. MemoryRead: در این حالت، واحد کنترل داده مورد نظر را از حافظه خوانده و به رجیستر مورد نظر اختصاص می‌دهد و به حالت IncrementPC منتقل می‌شود.

۱۱. Store: در این حالت، واحد کنترل برای نوشتن داده از رجیستر مورد نظر به حافظه، به حالت MemoryWrite منتقل می‌شود.

۱۲. MemoryWrite: در این حالت، واحد کنترل داده مورد نظر را از رجیستر مورد نظر به حافظه می‌نویسد و به حالت IncrementPC منتقل می‌شود.

۱۳. Register-Immediate: در این حالت، واحد کنترل برای اجرای دستوراتی که شامل یک رجیستر و یک مقدار ثابت هستند، به حالت Immediate۱ منتقل می‌شود.

۱۴. Immediate۱: در این حالت، واحد کنترل مقدار ثابت را از دستور خوانده و به حالت Immediate۲ منتقل می‌شود.

۱۵. Immediate۲: در این حالت، واحد کنترل سیگنال‌های کنترلی برای اجزای مختلف پردازنده را تولید می‌کند و به حالت ExecuteImmediate منتقل می‌شود.

۱۶. ExecuteImmediate: در این حالت، دستور با مقدار ثابت در حال اجرا است و واحد کنترل سیگنال‌های کنترلی برای پردازش دستور، به حالت IncrementPC منتقل می‌شود.

کد طراحی شده برای این ماژول در صفحات بعد قرار گرفته است.

```
entity ControlUnit is
    Port ( clk             : in  STD_LOGIC;
           rst             : in  STD_LOGIC;
           Op              : in  STD_LOGIC_VECTOR (5 downto 0);
           Func            : in  STD_LOGIC_VECTOR (5 downto 0);
           PCWriteCond_beq : out std_logic;
           PCWriteCond_bne : out std_logic;
           PCWrite         : out std_logic;
           IorD            : out std_logic;
           MemRead         : out std_logic;
           MemWrite        : out std_logic;
           MemToReg        : out std_logic_vector(1 downto 0);
           IRWrite         : out std_logic;
           PCSource        : out std_logic_vector(1 downto 0);
           ALUOp           : out std_logic_vector(2 downto 0);
           ALUSrcB         : out std_logic_vector(1 downto 0);
           ALUSrcA         : out std_logic;
           RegWrite        : out std_logic;
           RegDst          : out std_logic_vector(1 downto 0));
end ControlUnit;

architecture Behavioral of ControlUnit is

type state is(   Start,
                 InstructionFetch,
                 InstructionFetch1,
                 InstructionFetch2,
                 InstructionDecode,
                 MemoryAddressComp,
                 JR_Execution,
                 JAL_Execution,
                 JAL_Completion,
                 JALR_Execution,
                 JALR_Completion,
                 Execution,
                 Execution_addi,
                 Execution_slti,
                 Execution_andi,
                 Execution_ori,
                 Execution_xori,
                 Execution_lui,
                 BranchCompletion_beq,
                 BranchCompletion_bne,
                 JumpCompletion,
                 MemoryAccessLoad,
                 MemoryAccessStore,
                 RTypeCompletion,
                 RTypeCompletion_I,
                 MemoryReadCompletionStep );

 signal current_state, next_state : state := Start;
```

```
 signal ctrl_state : std_logic_vector(19 downto 0) := (others => '0');

begin

process(clk, rst, Op)
  begin
    if rst = '0' then
      current_state <= Start;
    elsif rising_edge(clk) then
      current_state <= next_state;
    end if;

    case current_state is

      when Start             => next_state <= InstructionFetch;
      when InstructionFetch  => next_state <= InstructionFetch1;
      when InstructionFetch1  => next_state <= InstructionFetch2;
      when InstructionFetch2  => next_state <= InstructionDecode;

      when InstructionDecode => if Op = "100011" then -- lw
                                  next_state <= MemoryAddressComp;
                                elsif Op = "101011" then -- sw
                                  next_state <= MemoryAddressComp;
                                elsif Op = "000000" and Func = "001000" then -- R-type JR
                                  next_state <= JR_Execution;
                                elsif Op = "000000" and Func = "001001" then -- R-type JALR
                                  next_state <= JALR_Execution;
                                elsif Op = "000000" then -- R-type
                                    next_state <= Execution;

                                elsif Op = "001000" then -- addi
                                  next_state <= Execution_addi;
                                elsif Op = "001010" then -- slti
                                  next_state <= Execution_slti;
                                elsif Op = "001101" then -- ori
                                  next_state <= Execution_ori;
                                elsif Op = "001100" then -- andi
                                  next_state <= Execution_andi;
                                elsif Op = "001110" then -- xori
                                  next_state <= Execution_xori;
                                elsif Op = "001111" then -- lui
                                  next_state <= Execution_lui;

                                elsif Op = "000100" then -- BEQ
                                  next_state <= BranchCompletion_beq;
                                elsif Op = "000101" then -- BNE
                                  next_state <= BranchCompletion_bne;
                                elsif Op = "000010" then -- Jump
                                  next_state <= JumpCompletion;
                                elsif Op = "000011" then -- JAL
                                  next_state <= JAL_Execution;
                                end if;

      when MemoryAddressComp => if Op = "100011" then -- lw
                                  next_state <= MemoryAccessLoad;
                                else  -- sw
```

```
                        next_state <= MemoryAccessStore;
                    end if;

when JR_Execution          => next_state <= InstructionFetch;

when JAL_Execution         => next_state <= JAL_Completion;

when JAL_Completion        => next_state <= InstructionFetch;

when JALR_Execution        => next_state <= JALR_Completion;

when JALR_Completion       => next_state <= InstructionFetch;

when Execution             => next_state <= RTypeCompletion;

when Execution_addi        => next_state <= RTypeCompletion_I;

when Execution_slti        => next_state <= RTypeCompletion_I;

when Execution_andi        => next_state <= RTypeCompletion_I;

when Execution_ori         => next_state <= RTypeCompletion_I;

when Execution_xori        => next_state <= RTypeCompletion_I;

when Execution_lui         => next_state <= RTypeCompletion_I;

when MemoryAccessLoad      => next_state <= MemoryReadCompletionStep;
```

```
    when MemoryAccessStore     => next_state <= InstructionFetch;

    when BranchCompletion_beq  => next_state <= InstructionFetch;

    when BranchCompletion_bne  => next_state <= InstructionFetch;

    when JumpCompletion        => next_state <= InstructionFetch;

    when RTypeCompletion       => next_state <= InstructionFetch;

    when others                => next_state <= InstructionFetch;

  end case;
end process;

with current_state select
  ctrl_state <= "00000000000XXX000000" when Start,
                "00001000100000010000" when InstructionFetch,
                "00001000100000010000" when InstructionFetch1,
                "10001000100000010000" when InstructionFetch2,

                "00000000000000110000" when InstructionDecode,
                "00000000000000101000" when MemoryAddressComp,
                "00000000000010001000" when Execution,

                "00000000000000101000" when Execution_addi,
                "00000000000011101000" when Execution_slti,
                "00000000000100101000" when Execution_andi,
```

```
                    "00000000000101101000" when Execution_ori,
                    "00000000000001101000" when Execution_xori,
                    "00000000000110101000" when Execution_lui,

                    "00100000001001001000" when BranchCompletion_beq,

                    "01000000001001001000" when BranchCompletion_bne,

                    "10000000011000000000" when JR_Execution,
                    "10000000010000000000" when JumpCompletion,

                    "00000010000000110010" when JAL_Execution,
                    "10000100100000110110" when JAL_Completion,

                    "00000010000000110001" when JALR_Execution,
                    "10000100110000110101" when JALR_Completion,

                    "00011000000000000000" when MemoryAccessLoad,
                    "00010100000000000000" when MemoryAccessStore,
                    "00000000000000000101" when RTypeCompletion,
                    "00000000000000000100" when RTypeCompletion_I,
                    "00000001000000000100" when MemoryReadCompletionStep,
                    "00000000000000000000" when others;

  PCWrite          <= ctrl_state(19);
  PCWriteCond_bne <= ctrl_state(18);
  PCWriteCond_beq <= ctrl_state(17);
  IorD             <= ctrl_state(16);
                      _
  MemRead          <= ctrl_state(15);
  MemWrite         <= ctrl_state(14);
  MemToReg         <= ctrl_state(13 downto 12);
  IRWrite          <= ctrl_state(11);
  PCSource         <= ctrl_state(10 downto 9);
  ALUOp            <= ctrl_state(8 downto 6);
  ALUSrcB          <= ctrl_state(5 downto 4);
  ALUSrcA          <= ctrl_state(3);
  RegWrite         <= ctrl_state(2);
  RegDst           <= ctrl_state(1 downto 0);

end Behavioral;
```

فصل ششم

پیاده سازی PC و Sign Extend

در این فصل قصد داریم نحوه کارکرد و شیوه پیاده سازی ماژول های PC و Sign Extend را شرح دهیم.

PC ۱-۶

PC برای نگهداری و محاسبه آدرس برنامه استفاده می‌شود. آدرس برنامه شامل آدرس حافظه‌ای است که در آن دستورات برنامه قرار دارند. با استفاده از این ماژول، می‌توان آدرس فعلی برنامه را در هر لحظه محاسبه کرد و در مراحل بعدی اجرای برنامه، از آن استفاده کرد.

با استفاده از کد VHDL یک ماژول PC را پیاده‌سازی کردیم که شامل یک رجیستر ۳۲ بیتی است و در هر لبه مثبت کلاک، آدرس فعلی برنامه را دریافت کرده و در خروجی خود قرار می‌دهد.

ورودی:

- CLK: کلاک سیستم
- EN: سیگنالی که نشان می‌دهد آیا امکان تغییر در رجیستر وجود دارد یا خیر.
- PC_input: داده‌ای که برای نوشتن در رجیستر ورودی استفاده می‌شود.
- RST: سیگنال ریست که در صورت فعال شدن، مقدار رجیستر را صفر می‌شود.

خروجی:

- PC_output: آدرس فعلی برنامه که در رجیستر ذخیره شده است.

در طراحی این ماژول، از یک رجیستر ۳۲ بیتی به نام PC_output استفاده شده است. این رجیستر، آدرس فعلی برنامه را نشان می‌دهد که در هر لبه مثبت کلاک، داده‌های ورودی را دریافت کرده و در خود ذخیره می‌کند.

در فرآیند اجرای کد، اگر سیگنال RST فعال شده باشد، مقدار رجیستر را صفر می‌کنیم. در غیر این صورت، اگر کلاک رو به بالا باشد و سیگنال EN فعال باشد، آدرس جدیدی که در PC_input قرار دارد را در رجیستر ذخیره می‌کنیم. به این ترتیب، آدرس فعلی برنامه به مقدار جدید تغییر می‌کند.

کد مورد استفاده برای طراحی این ماژول را میتوان در ادامه مشاهده کرد:

```
entity PC is
    Port ( CLK : in  STD_LOGIC;
           EN : in  STD_LOGIC;
           PC_input : in  STD_LOGIC_VECTOR(31 downto 0);
           RST: in  STD_LOGIC;
           PC_output : out  STD_LOGIC_VECTOR(31 downto 0)
           );
end PC;

architecture Behavioral of PC is

begin

   process(CLK)
   begin
      if(RST = '0') then
            PC_output <= (others => '0');
      elsif(rising_edge(CLK)) then
         if(EN = '1') then
            PC_output <= PC_input;
         end if;
      end if;
   end process;

end Behavioral;
```

Sign Extend ۲-۶

این ماژول معمولاً در طراحی و پیاده‌سازی الگوریتم‌های مختلف در پردازشگرهای داده‌های ۳۲ بیتی استفاده می‌شود. برای مثال، در پردازشگرهایی که در آنها از محاسبات با عددهای با علامت استفاده می‌شود، این ماژول برای تبدیل عددهای ورودی به فرمتی با علامت ۳۲ بیتی مورد استفاده قرار می‌گیرد. به عنوان مثال، ممکن است در محاسبات مربوط به مسائل حسابی، بخشی از داده‌ها به صورت عدد صحیح با علامت ۱۶ بیتی باشد. در چنین مواردی، این ماژول برای تبدیل داده‌های ورودی به فرمتی با علامت ۳۲ بیتی مورد استفاده قرار می‌گیرد.

در پردازنده MIPS نیز از این ماژول استفاده میشود. این ماژول شامل یک فرآیند است که با توجه به بیت علامتی عدد ورودی، این عدد را به صورت یک عدد با علامت ۳۲ بیتی گسترش می‌دهد.

ورودی:

- Input: عدد ۱۶ بیتی که می‌خواهیم به یک عدد با علامت ۳۲ بیتی تبدیل کنیم.

خروجی:

- Output: عدد ۳۲ بیتی با علامتی که در نهایت به دست می‌آید.

در طراحی این ماژول، از یک فرآیند استفاده شده است. در این فرآیند، بر اساس بیت علامت عدد ورودی، ابتدا بیت‌های ۱۶ تا ۰ عدد ورودی به خروجی منتقل می‌شوند. سپس، در صورتی که بیت علامتی عدد ورودی ۱ باشد، بیت‌های ۳۱ تا ۱۶ خروجی را با ۱ پر می‌کنیم تا عدد تبدیل شده با علامت منفی شود. در غیر این صورت، بیت‌های ۳۱ تا ۱۶ خروجی را با ۰ پر می‌کنیم تا عدد تبدیل شده با علامت مثبت باشد.

کدهای نوشته شده برای پیاده سازی آن را میتوانید در ادامه مشاهده کنید:

```
entity Sign_Extended is
    Port ( input : in  STD_LOGIC_VECTOR (15 downto 0);
           output : out  STD_LOGIC_VECTOR (31 downto 0));
end Sign_Extended;

architecture Behavioral of Sign_Extended is

begin

   process(input)
   begin
      if input(15)='0' then
         output(15 downto 0) <= input;
         output(31 downto 16) <= "0000000000000000";
      else
         output(15 downto 0) <= input;
         output(31 downto 16) <= "1111111111111111";
      end if;
   end process;

end Behavioral;
```

فصل هفتم

Top Module

در این بخش اجزا و ماژول‌های طراحی شده قبل را به هم مرتبط میکنیم.

```
entity MIPS_Processor is
    Port ( CLKmain, RSTMain : in  STD_LOGIC;
           output_top : out STD_LOGIC);
end MIPS_Processor;

architecture Behavioral of MIPS_Processor is

COMPONENT PC
    Port ( CLK        : in  STD_LOGIC;
           EN         : in  STD_LOGIC;
           PC_input   : in  STD_LOGIC_VECTOR(31 downto 0);
           RST        : in  STD_LOGIC;
           PC_output  : out  STD_LOGIC_VECTOR(31 downto 0)
           );
end COMPONENT;

COMPONENT MUX2to1
    Port ( a          : in  STD_LOGIC_VECTOR (31 downto 0);
           b          : in  STD_LOGIC_VECTOR (31 downto 0);
           sel        : in  STD_LOGIC;
           output     : out STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;
```

```
COMPONENT MUX3to1
    Port ( a          : in  STD_LOGIC_VECTOR (31 downto 0);
           b          : in  STD_LOGIC_VECTOR (31 downto 0);
           c          : in  STD_LOGIC_VECTOR (31 downto 0);
           sel        : in  STD_LOGIC_VECTOR (1 downto 0);
           output     : out STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;

COMPONENT MUX4to1
    Port ( a          : in  STD_LOGIC_VECTOR (31 downto 0);
           b          : in  STD_LOGIC_VECTOR (31 downto 0);
           c          : in  STD_LOGIC_VECTOR (31 downto 0);
           d          : in  STD_LOGIC_VECTOR (31 downto 0);
           sel        : in  STD_LOGIC_VECTOR (1 downto 0);
           output     : out STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;

COMPONENT MUX3to1_RegDst
    Port ( a          : in  STD_LOGIC_VECTOR (4 downto 0);
           b          : in  STD_LOGIC_VECTOR (4 downto 0);
           c          : in  STD_LOGIC_VECTOR (4 downto 0);
           sel        : in  STD_LOGIC_VECTOR (1 downto 0);
           output     : out STD_LOGIC_VECTOR (4 downto 0));
end COMPONENT;

COMPONENT ShiftLeft2_32
    Port ( input      : in  STD_LOGIC_VECTOR (31 downto 0);
                                 _          _
           output     : out STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;

COMPONENT ShiftLeft2_26
    Port ( input      : in  STD_LOGIC_VECTOR (25 downto 0);
           output     : out STD_LOGIC_VECTOR (27 downto 0));
end COMPONENT;

COMPONENT Sign_Extended
    Port ( input      : in  STD_LOGIC_VECTOR (15 downto 0);
           output     : out STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;

COMPONENT Memory
    Port ( memory_read  : in  STD_LOGIC;
           memory_write : in  STD_LOGIC;
           address      : in  STD_LOGIC_VECTOR (31 downto 0);
           write_data   : in  STD_LOGIC_VECTOR (31 downto 0);
           clk          : in  STD_LOGIC;
           rst          : in  STD_LOGIC;
           memory_data  : out STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;
```

```
COMPONENT InstructionRegister
    Port ( clk                : in  STD_LOGIC;
           rst                : in  STD_LOGIC;
           IRWrite            : in  STD_LOGIC;
           input_instruction  : in  STD_LOGIC_VECTOR (31 downto 0);
           output_instruction : out  STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;

COMPONENT MemoryDataRegister
    Port ( clk                : in  STD_LOGIC;
           rst                : in  STD_LOGIC;
           mem_input          : in  STD_LOGIC_VECTOR (31 downto 0);
           mem_output         : out  STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;

COMPONENT Registers
    Port ( clk                : in  STD_LOGIC;
           rst                : in  STD_LOGIC;
           read_register1     : in  STD_LOGIC_VECTOR (4 downto 0);
           read_register2     : in  STD_LOGIC_VECTOR (4 downto 0);
           write_register     : in  STD_LOGIC_VECTOR (4 downto 0);
           write_data         : in  STD_LOGIC_VECTOR (31 downto 0);
           RegWrite           : in  STD_LOGIC;
           read_data1         : out  STD_LOGIC_VECTOR (31 downto 0);
           read_data2         : out  STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;
```

```
COMPONENT ALU
    Port ( a            : in  STD_LOGIC_VECTOR (31 downto 0);
           b            : in  STD_LOGIC_VECTOR (31 downto 0);
           ALU_control  : in  STD_LOGIC_VECTOR (3 downto 0);
           output       : out  STD_LOGIC_VECTOR (31 downto 0);
           zero         : out  STD_LOGIC);
end COMPONENT;

COMPONENT ALUControl
    Port ( instruction  : in  STD_LOGIC_VECTOR (5 downto 0);
           ALUOp        : in  STD_LOGIC_VECTOR (2 downto 0);
           output       : out  STD_LOGIC_VECTOR (3 downto 0));
end COMPONENT;

COMPONENT ControlUnit
    Port ( clk             : in  STD_LOGIC;
           rst             : in  STD_LOGIC;
           Op              : in  STD_LOGIC_VECTOR (5 downto 0);
           Func            : in  STD_LOGIC_VECTOR (5 downto 0);
           PCWriteCond_beq : out std_logic;
           PCWriteCond_bne : out std_logic;
           PCWrite         : out std_logic;
           IorD            : out std_logic;
           MemRead         : out std_logic;
           MemWrite        : out std_logic;
           MemToReg        : out std_logic_vector(1 downto 0);
           IRWrite         : out std_logic;
           PCSource        : out std_logic_vector(1 downto 0);
           ALUOp           : out std_logic_vector(2 downto 0);
           ALUSrcB         : out std_logic_vector(1 downto 0);
           ALUSrcA         : out std_logic;
           RegWrite        : out std_logic;
           RegDst          : out std_logic_vector(1 downto 0));
end COMPONENT;

COMPONENT TempRegister
    Port ( clk       : in  STD_LOGIC;
           rst       : in  STD_LOGIC;
           input     : in  STD_LOGIC_VECTOR (31 downto 0);
           output    : out  STD_LOGIC_VECTOR (31 downto 0));
end COMPONENT;

--constant
constant PC_increment : std_logic_vector(31 downto 0) := "00000000000000000000000000000100";
constant Register_31 : std_logic_vector(4 downto 0) := "11111"; --31

-- signals
signal MUX_to_PC, PC_out, ALUOut_to_MUX, MUX_to_Memory, RegB_out, Memory_out, Instruction_Reg
       Memory_Data_Register_out, MUX3_to_Register, Registers_to_A, Registers_to_B, Sign_Exter
       Shift_Left2_32_to_MUX5, RegA_to_MUX, MUX4_to_ALU, MUX5_to_ALU,
       Alu_Out, Jump_Address                : std_logic_vector(31 downto 0);

signal MUX2_to_Register                     : std_logic_vector(4 downto 0);

signal ALUControl_to_ALU                    : std_logic_vector(3 downto 0);
```

```
signal ALUOp                                    : std_logic_vector(2 downto 0);

signal PCsource, ALUSrcB, MemToReg, RegDst      : std_logic_vector(1 downto 0);

signal IorD, MemRead, MemWrite, IRWrite, RegWrite, ALU_Zero, PCWriteCond_beq,
       PCWriteCond_bne, PCWrite, ALUSrcA,
       AND_to_OR2, AND_to_OR1, OR_to_PC         : std_logic;

begin

   AND_to_OR1 <= ALU_Zero and PCWriteCond_beq;
   AND_to_OR2 <= (not ALU_Zero) and PCWriteCond_bne;
   OR_to_PC <= AND_to_OR1 or PCWrite or AND_to_OR2;
   Jump_Address(31 downto 28) <= PC_out(31 downto 28);


   ALU_inst          : ALU                    port map(MUX4_to_ALU, MUX5_to_ALU, ALUControl_to
   ALU_Control       : ALUControl             port map(Instruction_Register_out(5 downto 0), A
   Control_Unit      : ControlUnit            port map(CLKMain, RSTMain, Instruction_Register_
   Inst_Reg          : InstructionRegister    port map(CLKMain, RSTMain, IRWrite, Memory_out,
   Memory_inst       : Memory                 port map(MemRead, MemWrite, MUX_to_Memory, RegB_
   MEM_DATA_REG      : MemoryDataRegister     port map(CLKMain, RSTMain, Memory_out, Memory_Da
   MUX_1             : MUX2to1                port map(PC_out, ALUOut_to_MUX, IorD, MUX_to_Mem
   MUX_2             : MUX3to1_RegDst         port map(Instruction_Register_out(20 downto 16),
   MUX_3             : MUX3to1                port map(ALUOut_to_MUX, Memory_Data_Register_out
   MUX_4             : MUX2to1                port map(PC_out, RegA_to_MUX, ALUSrcA, MUX4_to_A
   MUX_5             : MUX4to1                port map(RegB_out, PC_increment, Sign_Extend_out

   MUX_6             : MUX4to1               port map(Alu_Out, ALUOut_to_MUX, Jump_Address, Re
   Program_Counter   : PC                    port map(CLKMain, OR_to_PC, MUX_to_PC, RSTMain, P
   Registers_inst    : Registers             port map(CLKMain, RSTMain, Instruction_Register_c
   Sign_Extend       : Sign_Extended         port map(Instruction_Register_out(15 downto 0), S
   Shift_Left2_1     : ShiftLeft2_32         port map(Sign_Extend_out, Shift_Left2_32_to_MUX5)
   Shift_Left2_2     : ShiftLeft2_26         port map(Instruction_Register_out(25 downto 0), J
   Reg_A             : TempRegister          port map(CLKMain, RSTMain, Registers_to_A, RegA_t
   Reg_B             : TempRegister          port map(CLKMain, RSTMain, Registers_to_B, RegB_c
   Reg_ALUOut        : TempRegister          port map(CLKMain, RSTMain, Alu_Out, ALUOut_to_MUX

   output_top <= AND_to_OR1;
end Behavioral;
```

در واقع با کدهای بحث شده یک پردازنده ساده با معماری MIPS را پیاده‌سازی می‌کند که قادر است دستورات ساده‌ای را اجرا کند. این پردازنده از چندین قسمت مهم تشکیل شده است:

قسمت اول این پردازنده، شمارنده برنامه یا PC می‌باشد که مسئولیت نگهداری آدرس دستور فعلی را بر عهده دارد. آدرس دستور فعلی در این قسمت ذخیره می‌شود و پس از اجرای هر دستور، به طور خودکار ۴ واحد به آن افزوده می‌شود تا آدرس دستور بعدیاجرا شود.

قسمت دوم این پردازنده، رجیسترها هستند. این قسمت شامل ۳۲ رجیستر مختلف است که هر کدام ۳۲ بیتی هستند. این رجیسترها برای نگهداری مقادیر ورودی و خروجی دستورات استفاده می‌شوند.

سومین قسمت، واحد منطقی حساب و ریاضی یا ALU است که مسئولیت اجرای عملیات‌های محاسباتی بین دو عدد ورودی (مانند جمع، تفریق، ضرب و تقسیم) را بر عهده دارد. ALU به دو عدد ورودی، یک عملگر ورودی و یک علامت گیرنده ورودی نیاز دارد و نتیجه را به عنوان خروجی ارائه می‌کند.

قسمت چهارم، واحد کنترل است که مسئولیت کنترل و تنظیم عملکرد پردازش را بر عهده دارد. این قسمت، با توجه به دستورات ورودی، دستورات لازم را به بخش‌های مختلف پردازنده می‌دهد تا دستورات اجرا شوند. این واحد از سیگنال‌های مختلفی برای کنترل پردازنده استفاده می‌کند که به آن‌ها اجازه می‌دهد که اجرای دستورات را تنظیم کنند.

قسمت آخر، حافظه است که مسئولیت ذخیره داده‌ها و دستورات را بر عهده دارد. این حافظه دارای یک آدرس‌دهی است که با استفاده از آن، داده‌ها و دستورات مورد نیاز را از حافظه خوانده و به آن‌ها نوشته می‌شود. همچنین، این حافظه به عنوان محلی برای ذخیره‌سازی داده‌های موقتی استفاده می‌شود که در طول اجرای دستورات، نیاز به آن‌ها پیش می‌آید.

برای اجرای دستورات، ابتدا آدرس دستور فعلی توسط شمارنده برنامه بارگذاری می‌شود و سپس دستور با استفاده از واحد ثبت دستور از حافظه خوانده شده و در داخل رجیسترها ذخیره می‌شود. سپس، با استفاده از واحد کنترل، دستورات لازم به قسم‌های مختلف پردازنده داده می‌شود تا دستورات اجرا شوند. در هنگام اجرای دستور، ابتدا مقادیر مورد نیاز از رجیسترها یا حافظه خوانده می‌شوند و سپس با استفاده از واحد ALU، عملیات‌های محاسباتی انجام می‌شوند. نتیجه عملیات در نهایت در رجیسترها یا حافظه ذخیره می‌شود. پس از اجرای دستور، شمارنده برنامه به طور خودکار به آدرس دستور بعدی افزایش پیدا می‌کند و پردازنده آماده اجرای دستور بعدی می‌شود.

منابع

۱. D.Harris , S.Harris. *Digital Design and Computer Architecture*
۲. M.Movahedin *MIPS Multi-Cycle Implentation*
۳. O.Chakon *Design, Modeling, and Simulation of a MIPS Multi-Cycle Processor*
۴. Patterson, D.A. and J.L. Hennesey. *Computer Organization and Design: The Hardware/Software Interface*
۵. https://www.youtube.com/watch?app=desktop&v=hmK3xabU6Ts
۶. https://github.com/etchsaleh/MultiCycleMIPS