

گزارشکار پروژه طراحی کامپیوتری سیستم‌های دیجیتال

اعضاي گروه: نگين صصاصي ، رسول باقرى

موضوع پروژه: طراحی و پیاده‌سازی پردازنده MIPS (Multi-Cycle)

پیاده‌سازی کامل مسیر داده (Data Path) و واحد کنترل (Control Unit) یک پردازنده MIPS چند مرحله‌ای به همراه شبیه‌سازی و تست سخت‌افزارهای داخلی شامل رجیسترها، مالتی‌پلکسرهای ALU و حافظه، با استفاده از زبان توصیف سخت‌افزار Verilog و تحلیل عملکرد در نرم‌افزار شبیه‌ساز

Data Path :

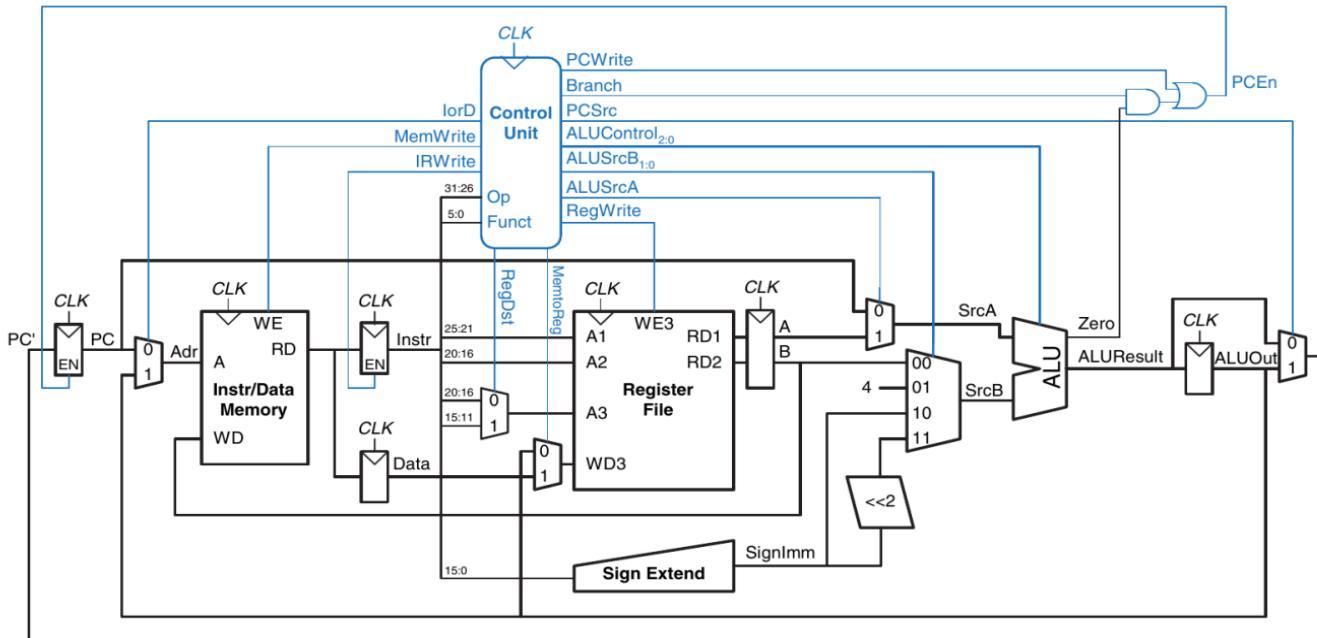


Figure 7.27 Complete multicycle MIPS processor

در معماری multicycle ، هر دستور در چند سیکل کلاک اجرا می شود و سخت افزارهای مشترک در سیکل های مختلف باز استفاده می شوند. برای این کار، مسیر داده دارای چند رजیستر موقتی و مالتی پلکسor است تا مقادیر بین مراحل حفظ و انتخاب شوند. اجزای اصلی:

اجزای کلیدی:

- **PC (Program Counter)** : آدرس دستور فعلی با سیگنال **PCEn** به روزرسانی می شود.
- **Memory (Instr/Data)** : حافظه مشترک دستور/داده.
- **IorD** : (انتخاب آدرس : PC یا PC') (نوشتن) **MemWrite** ، (بارگذاری دستور) **IRWrite** ، (نگهداری دستور) **MemData** .
- **IR (Instruction Register)** : نگهداری دستور خوانده شده از حافظه.
- **MDR (Memory Data Register)** : نگهداری داده خوانده شده از حافظه تا سیکل بعد.
- **Register File** : رجیستر های معماری.
- **RegWrite** : (انتخاب مقصد : ALUOut/MDR) (منبع نوشت : rt/rd) ، **MemtoReg** ، (ALUOut/MDR) .
- **A و B** : رجیستر های موقتی خروجی خواندن رجیستر فایل (ورودی های ALU را یک سیکل نگه می دارند).
- **SignImm<2** و **SignExtend** : تولید **SignImm** و نسخه ای شیفت خورده ای آن برای محاسبات شاخه / آدرس مؤثر.
- **MUX** : **RegDst** ، **MemtoReg** ، **ALUSrcA** ، **ALUSrcB[1:0]** ، **IorD** ، **PCEn** .
- **ALUOut** : انتخاب ورودی A از ALU بین PC و رجیستر A .
- **ALUSrcA** : انتخاب ورودی B از ALU بین B ، ثابت 4 و SignImm<2 .
- **ALUSrcB[1:0]** : انتخاب منبع آدرس حافظه (ALUOut یا PC) .
- **IorD** : انتخاب رجیستر مقصد و داده هی نوشتن در رجیستر فایل .
- **PCEn** : انتخاب رجیستر مقصد و داده هی نوشتن در رجیستر فایل .

(ALUOut و ALUResult بین PC انتخاب منبع ورودی) PCSrc .

. Zero و ALUResult کنترل ALU و تولید ALUControl[2:0] : انجام عملیات منطقی/حسابی با .

. ALUOut رجیستر نگهدارندهی خروجی ALU برای استفاده در سیکل های بعد (Mثلاً آدرس مؤثر یا نتیجه R-type) . OR (PCWrite) (Branch & Zero) از PCEn از می شود.

جريان داده در مراحل (چرخه های اجرای دستور)

Instruction Fetch (IF)-1

. PC = آدرس حافظه → IorD=0 .
دستور از حافظه به IR می روید .
ALU → (PC + 4) مقدار ALUSrcB=01 و ALUSrcA=0 (ثابت 4) انتخاب PC را می سازد .
PCEn , PC ← PC+4 → PCWrite=1 و PCSrc=0 با فعال شدن .

Instruction Decode / Register Fetch (ID) -2

. خواندن RS و RT از Register File و ذخیره در A و B .
SignExtend تولید SignImm می کند و SignImm<<2 نیز آماده می شود .
ALUSrcB=11 (SignImm<<2) → ALUResult و ALUSrcA=0 (PC) هم زمان، برای آدرس شاخه (SignImm<<2) → ALUResult نگه داشته می شود .
شاخه احتمالی را می سازد و در ALUOut / یا ALUOut نگه داشته می شود .

Execute / Address Calculation (EX) -3

. funct از R-type: ALUSrcB=00 (B) , ALUSrcA=1 (A) , ALUControl .
Load/Store: ALUSrcA=1 , ALUSrcB=10 (SignImm) → ALUOut = A + SignImm .
Branch (BEQ): ALUSrcA=1 , ALUSrcB=00 → ALU و Zero=1 تفريقي می کند؛ اگر Branch=1 آنگاه PCEn فعال و PCSrc منبع شاخه را انتخاب می کند (معمولاً ALUOut که در ID محاسبه شده) .

Memory Access (MEM) -4

. (ALUOut از آدرس) MemWrite=1 → MDR نوشتمن مقدار B در حافظه .
LW: IorD=1 → خواندن از حافظه و ذخیره در MDR .

Write Back (WB)-5

.RegWrite=1 و ALUOut از R-type: RegDst=1 (rd), MemtoReg=0 .

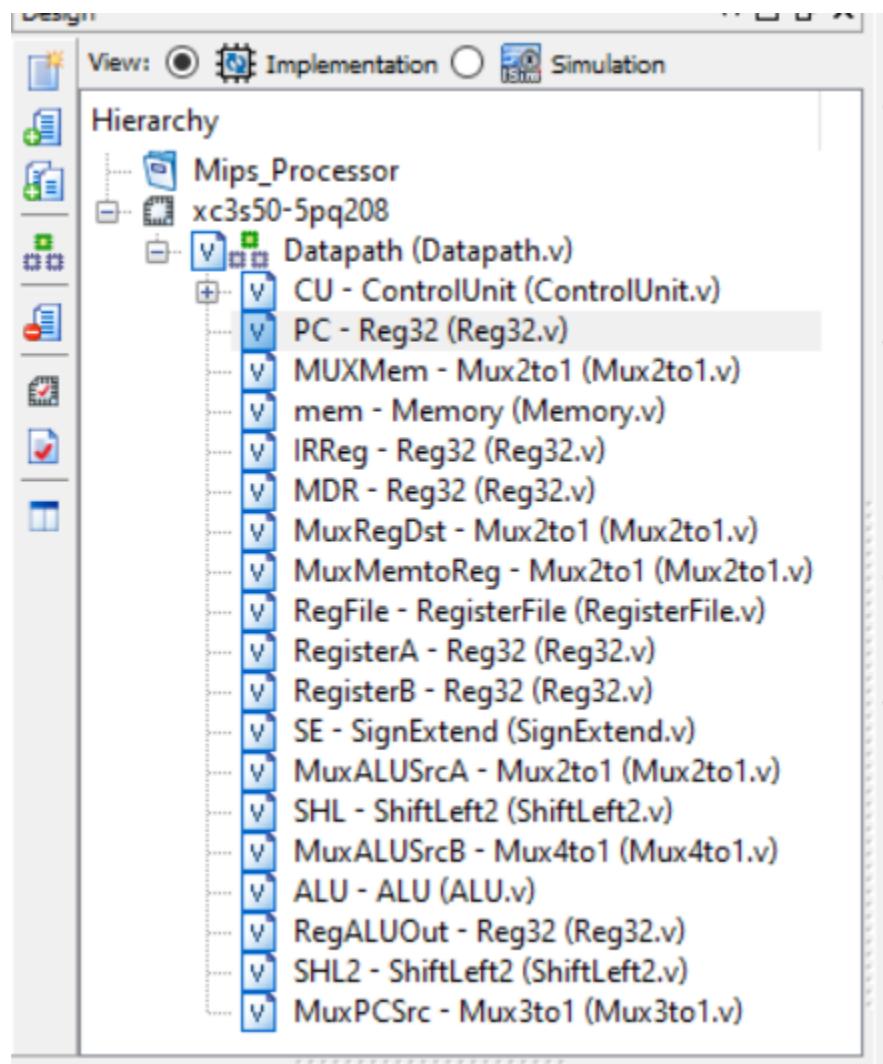
.RegWrite=1 (MDR) و LW: RegDst=0 (rt), MemtoReg=1 .

• با چند سیکل کردن اجرا، یک ALU و یک Memory برای همه مراحل مشترک می‌مانند \Rightarrow هزینه سخت‌افزار کمتر.

• رجیسترها میانی (IR, MDR, A, B, ALUOut) داده‌ها را بین سیکل‌ها نگه می‌دارند تا با استفاده از منابع ممکن شود.

• سیگنال‌های کنترلی از Control Unit (FSM) می‌آیند و هر سیکل مسیر داده را برای همان مرحله تنظیم می‌کنند.

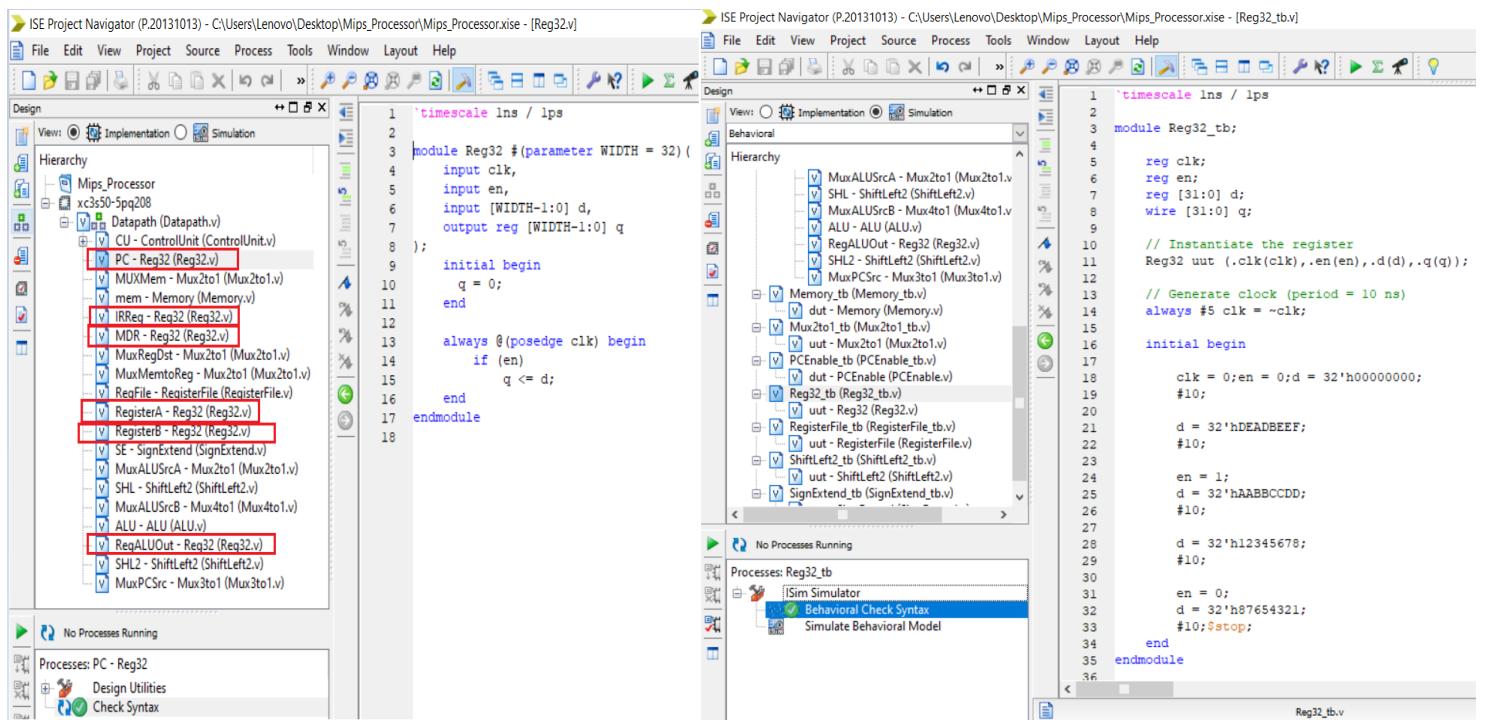
شمای کلی از مازول‌های استفاده شده در datapath



: datapath و توضیحات مازول‌های استفاده شده در

نام مازول: Reg32

نوع: ثبات (Register) با ورودی (Register)



این مازول در معماری پردازنده MIPS MultiCycle نقش یک Register عمومی ۳۲ بیتی را دارد.

عملکرد کلی:

- این مازول یک رجیستر لبه بالارونده (posedge clock) است.
- فقط زمانی که سیگنال en برابر 1 باشد، داده‌ی ورودی d در رجیستر ذخیره می‌شود.
- در حالت اولیه (initial) مقدار خروجی q صفر می‌شود تا پردازنده با حالت مشخص شروع کند.

ورودی‌ها و خروجی‌ها:

- clk: کلک – هر تغییر روی لبه‌ی بالارونده اعمال می‌شود.
- en: فعال‌ساز – (Enable) تعیین می‌کند آیا رجیستر مقدار جدید بگیرد یا نه.
- d: داده ورودی – مقداری که باید در رجیستر ذخیره شود.
- q: داده خروجی – مقدار فعلی ذخیره شده در رجیستر.

نقش در Data Path :

• چنین رجیستری می‌تواند برای **MDR (Memory Data Register)**، **IR (Instruction Register)**، **ALUOut** (خروجی فایل رجیسترها) یا **Register** به کار رود.

• چون این رجیستر پارامتریک است، می‌شود در هر نقطه از مسیر داده که به رجیستر با پهنهای خاص نیاز داریم استفاده شود.

• سیگنال **en** توسط واحد کنترل تعیین می‌شود تا در سیکل مناسب مقدار رجیستر به روز شود.

ارتباط با مراحل اجرای دستور در MIPS چند سیکلی:

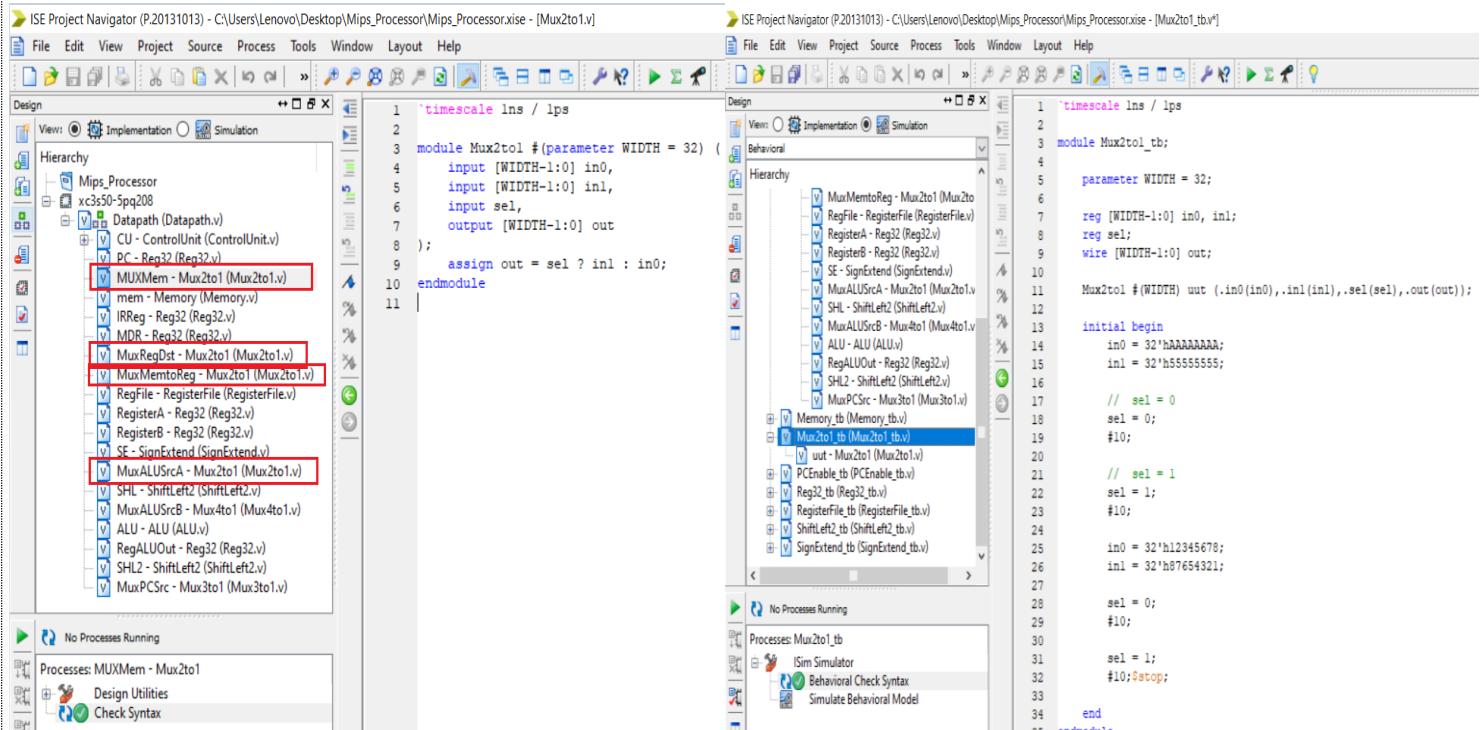
• رجیستر IR مقدار IR جدید دستور را از حافظه می‌گیرد.

• رجیسترها A و B داده خوانده شده از فایل رجیسترها را ذخیره می‌کنند.

• رجیستر EX/MEM/WB برای نگهداری نتایج موقت استفاده می‌شوند.

نام ماژول: Mux2to1

نوع: مالتی‌پلکسر دو به یک با پهنهای قابل تنظیم



این ماژول یک مالتی‌پلکسر دو به یک (2-to-1 MUX) پارامتریک است که در طراحی پردازنده MIPS MultiCycle انتخاب بین دو ورودی مختلف استفاده می‌شود.

عملکرد کلی:

- این مالتیپلکسر بین دو ورودی $in0$ و $in1$ انتخاب می‌کند.
- اگر سیگنال $sel = 0 \rightarrow$ خروجی برابر $in0$ خواهد بود.
- اگر سیگنال $sel = 1 \rightarrow$ خروجی برابر $in1$ خواهد بود.
- انتخاب به صورت ترکیبی (Combinational) انجام می‌شود و نیازی به کلاک ندارد.

ورودی‌ها و خروجی‌ها:

- $in0$: داده ورودی اول.
- $in1$: داده ورودی دوم.
- sel : سیگنال انتخاب – تعیین می‌کند کدام ورودی به خروجی وصل شود.
- out : خروجی انتخاب شده – یکی از دو ورودی به طور مستقیم در خروجی قرار می‌گیرد.

نقش در Data Path :

- این نوع MUX در چند نقطه از مسیر داده MIPS استفاده می‌شود:
 - انتخاب 4 یا PC+4 برای Target Address.
 - انتخاب بین خروجی ALU و داده حافظه برای نوشتن در فایل رجیستر (WB stage).
 - انتخاب بین Immediate و رجیستر B برای ورودی ALU.
- چون این مازول پارامتریک است، می‌تواند برای سیگنال‌های 32 بیتی، 5 بیتی یا هر پهنازی به کار رود.

ارتباط با مراحل اجرای دستور در MIPS چندسیکل:

- IF : انتخاب آدرس بعدی PC (معمولًاً بین PC+4 و آدرس شاخه).
- EX : انتخاب داده مناسب برای ورودی ALU.
- WB : انتخاب داده مناسب برای نوشتن در رجیستر مقصد.

نام ماژول : Memory

نوع : حافظه داده/Dستور ۳۲ بیتی با ۲۵۶ کلمه(word)

```

ISE Project Navigator (P.20131013) - C:\Users\Lenovo\Desktop\Mips_Processor\Mips_Processor.xise - [Memory.v]
ISE Project Navigator (P.20131013) - C:\Users\Lenovo\Desktop\Mips_Processor\Mips_Processor.xise - [Memory_tb.v]

Design View: Implementation Simulation
Design View: Implementation Simulation
Hierarchy Hierarchy
x3r50-5pq208 x3r50-5pq208
  Datapath (Datapath.v)
    CU - ControlUnit (ControlUnit.v)
    PC - Reg32 (Reg32.v)
    MUXMem - Mux2to1 (Mux2to1.v)
    mem - Memory (Memory.v)
    IRReg - Reg32 (Reg32.v)
    MDR - Reg32 (Reg32.v)
    MuxRegDst - Mux2to1 (Mux2to1.v)
    MuxMemToReg - Mux2to1 (Mux2to1.v)
    RegFile - Registerfile (Registerfile.v)
    RegisterA - Reg32 (Reg32.v)
    RegisterB - Reg32 (Reg32.v)
    SE - SignExtend (SignExtend.v)
    MuxALUSrcA - Mux2to1 (Mux2to1.v)
    SHL - ShiftLeft2 (ShiftLeft2.v)
    MuxALUSrcB - Mux4to1 (Mux4to1.v)
    ALU - ALU (ALU.v)
    RegALUOut - Reg32 (Reg32.v)
    SHL2 - ShiftLeft2 (ShiftLeft2.v)
    MuxPCSrc - Mux3to1 (Mux3to1.v)

Design View: Implementation Simulation
Design View: Implementation Simulation
Hierarchy Hierarchy
  MuxRegDst - Mux2to1 (Mux2to1.v)
  MuxMemToReg - Mux2to1 (Mux2to1.v)
  RegFile - Registerfile (Registerfile.v)
  RegisterA - Reg32 (Reg32.v)
  RegisterB - Reg32 (Reg32.v)
  SE - SignExtend (SignExtend.v)
  MuxALUSrcA - Mux2to1 (Mux2to1.v)
  SHL - ShiftLeft2 (ShiftLeft2.v)
  MuxALUSrcB - Mux4to1 (Mux4to1.v)
  ALU - ALU (ALU.v)
  RegALUOut - Reg32 (Reg32.v)
  SHL2 - ShiftLeft2 (ShiftLeft2.v)
  MuxPCSrc - Mux3to1 (Mux3to1.v)
  Memory_dut (
    .clk(clk),
    .MemWrite(MemWrite),
    .addr(addr),
    .write_data(write_data),
    .read_data(read_data)
  );
  always #5 clk = ~clk;
  initial begin
    clk = 0;
    MemWrite = 0;
    addr = 32'h00000000;
    write_data = 32'hDEADBEEF;
    // Ox08
    #10 MemWrite = 1;
    addr = 32'h00000008;
    write_data = 32'hDEADBEEF;
    // Ox08
    #10 MemWrite = 0; addr = 32'h00000008;
    #10 addr = 32'h0000000C;
    #10 $stop;
  end
endmodule

```

این ماژول حافظه اصلی پردازنده MIPS چندسیکل است. این حافظه هم برای دستورات (Instruction Memory) و هم برای داده‌ها (Data Memory) استفاده می‌شود.

عملکرد کلی:

- حافظه شامل ۲۵۶ خانه‌ی ۳۲ بیتی است (در مجموع ۱ کیلوبایت).
- در هنگام شبیه‌سازی، محتوای حافظه از یک فایل خارجی "assembler/init_mem.data" بارگذاری می‌شود.
- عملیات خواندن داده به صورت ترکیبی (بدون کلاک) انجام می‌شود.
- عملیات نوشتمن داده تنها در لبه‌ی بالارونده کلاک و در صورت فعال بودن MemWrite انجام می‌شود.
- آدرس‌دهی word-aligned است، یعنی از بیت‌های [۹:۰] آدرس برای انتخاب خانه حافظه استفاده می‌شود (چون هر خانه ۴ بایت است و دو بیت پایینی صفر هستند).

ورودی‌ها و خروجی‌ها:

ورودی‌ها:

- clk: سیگنال کلاک برای هماهنگی نوشتمن.

- :MemWrite باشد، داده در حافظه نوشته می‌شود.
- : آدرس ۳۲ بیتی (تنها بیت‌های ۹ تا ۲ استفاده می‌شوند).
- : داده‌ای که قرار است نوشته شود.
- خروجی:
- : داده‌ای که از حافظه خوانده می‌شود.

نقش در Data Path

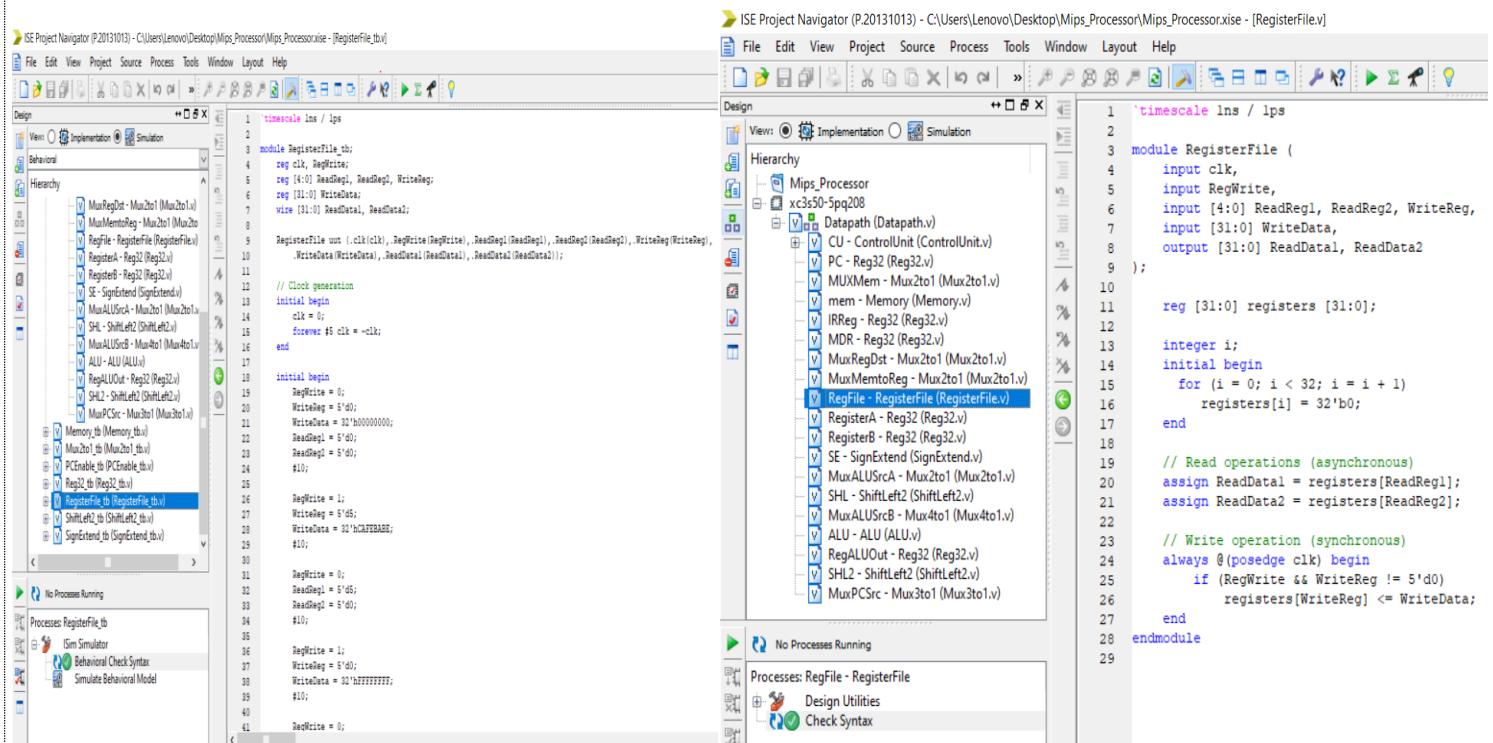
- در مرحله‌ی **IF (Instruction Fetch)**: پردازنده دستور را از همین حافظه می‌خواند.
- در مرحله‌ی **MEM (Memory Access)**: اگر دستور **lw** یا **sw** باشد، پردازنده داده را از همین حافظه می‌خواند یا در آن می‌نویسد.

ویژگی‌های مهم:

1. مقداردهی اولیه حافظه
 - با حلقه‌ی **for** تمام خانه‌های حافظه صفر می‌شوند.
 - سپس با **\$readmemb** مقادیر برنامه‌ی اسembly از فایل بارگذاری می‌شوند.
2. خواندن بدون کلاک و نوشتن با کلاک
 - **read_data** همیشه مقدار لحظه‌ای خانه‌ی حافظه انتخاب شده را بازتاب می‌دهد.
 - نوشتن تنها در لبه‌ی بالارونده کلاک و با **MemWrite = 1** انجام می‌شود.
3. آدرس‌دهی **Word-Aligned**
 - برای دسترسی به حافظه ۳۲ بیتی، دو بیت پایین آدرس (**addr[1:0]**) نادیده گرفته می‌شوند.

RegisterFile: مازول

نوع: فایل رجیستر هر کدام ۳۲x۳۲ (۳۲x۳۲ رجیستر)



این مازول Register File پردازنده MIPS است و دقیقاً مطابق ساختار ۳۲ رجیستری طراحی شده است.

عملکرد کلی:

- شامل ۳۲ رجیستر عمومی (R0 تا R31) هر کدام ۳۲ بیتی است.
- قابلیت خواندن همزمان از دو رجیستر و نوشتمن در یک رجیستر را دارد.
- رجیستر R0 همیشه صفر است و نمی‌توان در آن نوشت.
- خواندن داده‌ها غیرهمzman (asynchronous) انجام می‌شود.
- نوشتمن داده‌ها همزمان با کلاک (synchronous) و فقط وقتی RegWrite=1 باشد انجام می‌شود.

ورودی‌ها و خروجی‌ها:

- ورودی‌ها: clk: سیگنال کلاک برای کنترل نوشتمن
- : اگر ۱ باشد، داده در رجیستر مشخص شده نوشتمن می‌شود
- : آدرس رجیستری که باید روی پورت خواندن اول قرار گیرد
- : آدرس رجیستری که باید روی پورت خواندن دوم قرار گیرد

- آدرس رجیستری که باید در آن نوشته WriteReg
- دادهای که باید نوشته شود WriteData
- خروجی‌ها:

- مقدار رجیستر اول انتخاب شده ReadData1
- مقدار رجیستر دوم انتخاب شده ReadData2

نقش در Data Path

- در مرحله‌ی ID (Instruction Decode) شماره‌ی رجیسترهاخوانده شده از فیلد های rs و rt گرفته می‌شود و مقادیر آن‌ها به ترتیب به ReadData1 و ReadData2 ارسال می‌شود.
- در مرحله‌ی WB (Write Back) اگر دستور نیاز به نوشتمنتیجه داشته باشد (مثل lw , add , lw ، داده از طریق در رجیستر مقصد WriteReg) ذخیره می‌شود.

ویژگی‌های مهم:

- مقداردهی اولیه:
 - در ابتدای شبیه‌سازی، تمام ۳۲ رجیستر صفر می‌شوند.
- خواندن همزمان:
 - هر تغییری در آدرس‌های ورودی ReadReg1 یا ReadReg2 فوراً روی خروجی‌ها بازتاب می‌شود (بدون نیاز به کلک).
- نوشتمن همزمان و جلوگیری از تغییر R0:
 - فقط در لبه بالارونده کلک و وقتی $RegWrite=1$ فعال باشد انجام می‌شود.
 - دستور شرطی $WriteReg != 5'd0$ مانع از نوشتمن در رجیستر صفر می‌شود.

نام ماژول: SignExtend

نوع: توسعه‌دهنده‌ی علامت از ۱۶ بیت به ۳۲ بیت

```
'timescale 1ns / 1ps
module SignExtend (
    input [15:0] in,
    output [31:0] out
);
    assign out = {{16{in[15]}}, in};
endmodule
```

```
'timescale 1ns / 1ps
module SignExtend_tb;
    reg [15:0] in;
    wire [31:0] out;
    SignExtend uut (
        .in(in),
        .out(out)
    );
    initial begin
        in = 16'b0000_0000_0000_1010; // $10;
        in = 16'b1111_1111_1111_1111; // $FFFF;
        in = 16'b1111_1111_1000_0000; // $10;
        in = 16'b0111_1111_1111_1111; // $10;$stop;
    end
endmodule
```

این ماژول برای توسعه‌ی علامت (Sign Extension) استفاده می‌شود

عملکرد کلی:

- وظیفه‌ی این ماژول این است که عدد ۱۶ بیتی با علامت را به عدد ۳۲ بیتی تبدیل کند و علامت آن را حفظ کند.
- بیت پارازش (MSB) ورودی in[15] به عنوان بیت علامت در نظر گرفته می‌شود و در بیت‌های بالایی تکرار می‌شود.
- اگر $in[15] = 0$ → عدد مثبت بوده و ۱۶ بیت بالایی با صفر پر می‌شود.
- اگر $in[15] = 1$ → عدد منفی بوده و ۱۶ بیت بالایی با یک پر می‌شود (حفظ علامت).

ورودی‌ها و خروجی‌ها:

- ورودی: in [15:0] → عدد ۱۶ بیتی با علامت از دستوراتی مثل addi, lw, sw, beq

- خروجی: out [31:0] → نسخه‌ی ۳۲ بیتی همان عدد با حفظ علامت

نقش در Data Path :

- در پردازنده‌ی MIPS ، بیشتر دستورات Immediate 16 بیتی (addi, lw, sw, beq) دارای فیلد 16 بیتی immediate مثلاً MIPS هستند.

- این مازول مقدار ۱۶ بیتی را به ۳۲ بیتی تبدیل می‌کند تا بتواند مستقیماً وارد ALU یا سایر واحدها شود.

ویژگی‌های مهم:

۱. کاملاً ترکیبی (Combinational):

- هیچ کلاک یا لج ندارد، به محض تغییر ورودی خروجی به روزرسانی می‌شود.

۲. بدون تأخیر اضافی:

- صرفًاً اتصال سیمی است و از عملگر الحق { } برای پر کردن بیت‌های بالا استفاده شده.

نام مازول: ShiftLeft2

(Combinational Shifter)

```

ISE Project Navigator (P.20131013) - C:\Users\Lenovo\Desktop\Mips_Processor\Mips_Processor.xise - [ShiftLeft2.v]
ISE Project Navigator (P.20131013) - C:\Users\Lenovo\Desktop\Mips_Processor\Mips_Processor.xise - [ShiftLeft2_tb.v]

File Edit View Project Source Process Tools Window Layout Help
File Edit View Project Source Process Tools Window Layout Help

Design View: Implementation Simulation
Design View: Implementation Simulation

1 'timescale 1ns / 1ps
2
3 module ShiftLeft2 (
4   input [31:0] SignImm,
5   output [31:0] SignImmShifted
6 );
7   assign SignImmShifted = SignImm << 2;
8 endmodule
1 'timescale 1ns / 1ps
2
3 module ShiftLeft2_tb;
4
5   reg [31:0] SignImm;
6   wire [31:0] SignImmShifted;
7
8 // Instantiate the Unit Under Test (UUT)
9 ShiftLeft2 uut (.SignImm(SignImm),.SignImmShifted(SignImmShifted));
10
11 initial begin
12   SignImm = 32'h00000000;
13   $200;
14
15   SignImm = 32'd4;
16   $200;
17
18   SignImm = 32'h0000000F;
19   $200;
20
21   SignImm = 32'hFFFFFFFFFF;
22   $200;
23
24   SignImm = 32'h12345678;
25   $200;$stop;
26
27 end
28 endmodule

```

این مازول ShiftLeft2 در مسیر داده‌ی پردازنده‌ی MIPS برای شیفت دادن مقادیر Immediate به کار می‌رود.

عملکرد کلی:

- این مازول عدد ۳۲ بیتی ورودی را دو بیت به چپ شیفت می‌دهد (معادل ضرب در ۴).

• معمولاً برای محاسبه‌ی آدرس پرش‌ها یا شاخه‌ها (Branch) استفاده می‌شود، چون در MIPS آدرس‌ها Word-aligned هستند.

- انتقال تمام بیت‌های ورودی دو پله به چپ است.
- دو بیت کم ارزش صفر می‌شود.
- نتیجه در خروجی ظاهر می‌شود و هیچ تاخیر کلکی ندارد.

ورودی‌ها و خروجی‌ها:

- **ورودی:** SignExtend → معمولاً خروجی ماژول SignImm [31:0] ◦
- **خروجی:** SignImmShifted [31:0] ◦ مقدار ورودی در ۴ ضرب شده است.

نقش در Data Path :

- در دستورات شاخه (beq, bne) باید آفست ۱۶ بیتی به ۳۲ بیت گسترش یافته و سپس ۴ برابر شود چون آدرس دستورها به صورت word (۴ بایت) هستند.
- همچنین در محاسبه آدرس پرش‌های jump و jal نیز برای هم‌ترازی به کار می‌رود.

ویژگی‌های مهم:

1. **کاملاً ترکیبی (Combinational):**
 - بدون لج یا فلیپ‌فلاب، خروجی فوراً تغییر می‌کند.
2. **ساده و بهینه:**
 - فقط سیم‌کشی داخلی است و نیازی به منطق اضافی ندارد.

نام ماژول : Mux4to1

نوع : مالتی پلکسر ۴ به ۱

The screenshot shows the ISE Project Navigator interface. The top menu bar includes File, Edit, View, Project, Source, Process, Tools, Window, Layout, and Help. The main window has a toolbar at the top with various icons. On the left is a hierarchical tree view under 'Design' labeled 'Implementation'. It shows a project named 'xc3s50-5pq208' containing several sub-modules: Datapath (Datapath.v), CU - ControlUnit (ControlUnit.v), PC - Reg32 (Reg32.v), MUXMem - Mux2to1 (Mux2to1.v), mem - Memory (Memory.v), IRReg - Reg32 (Reg32.v), MDR - Reg32 (Reg32.v), MuxRegDst - Mux2to1 (Mux2to1.v), MuxMemtoReg - Mux2to1 (Mux2to1.v), RegFile - RegisterFile (RegisterFile.v), RegisterA - Reg32 (Reg32.v), RegisterB - Reg32 (Reg32.v), SE - SignExtend (SignExtend.v), MuxALUSrcA - Mux2to1 (Mux2to1.v), SHL - ShiftLeft2 (ShiftLeft2.v), MuxALUSrcB - Mux4to1 (Mux4to1.v), ALU - ALU (ALU.v), RegALUOut - Reg32 (Reg32.v), SHL2 - ShiftLeft2 (ShiftLeft2.v), and MuxPCSrc - Mux3to1 (Mux3to1.v). The right panel displays the Verilog code for the Mux4to1 module:

```
1 `timescale 1ns / 1ps
2
3 module Mux4to1 (
4     input [31:0] in0,
5     input [31:0] in1,
6     input [31:0] in2,
7     input [31:0] in3,
8     input [1:0] sel,
9     output reg [31:0] out
10 );
11     always @(*) begin
12         case (sel)
13             2'b00: out = in0;
14             2'b01: out = in1;
15             2'b10: out = in2;
16             2'b11: out = in3;
17             default: out = 32'b0;
18         endcase
19     end
20 endmodule
```

The bottom status bar indicates 'No Processes Running' and lists processes: 'Processes: MuxALUSrcB - Mux4to1' and 'Design Utilities' with 'Check Syntax' selected.

عملکرد کلی:

- این ماژول یکی از چهار ورودی ۳۲ بیتی را بر اساس سیگنال انتخاب sel به خروجی هدایت می‌کند.
- در پردازندۀ MIPS Multi-Cycle برای انتخاب منبع ورودی‌های ALU یا داده‌های نوشتني در رجیسترها به کار می‌رود.

ورودی‌ها و خروجی‌ها:

ورودی‌ها:

- داده‌های ۳۲ بیتی → in0, in1, in2, in3
- سیگنال انتخاب (کنترلی) → sel[1:0]

خروجی:

- یکی از ورودی‌ها بر اساس sel → out[31:0]

نقش در Data Path :

- در واحد ALU ، این مازول می‌تواند یکی از منابع مختلف را برای ورودی انتخاب کند.
- همچنین در مرحله Write-back رجیسترها، می‌تواند تعیین کند داده از حافظه، ALU یا PC+4 انتخاب شود.
- به صورت کلی هر جا که باید از بین چند مسیر مختلف فقط یکی انتخاب شود، این مازول ضروری است.

ویژگی‌های مهم:

1. کاملاً ترکیبی (بدون تاخیر کلاک) → نتیجه انتخاب فوراً روی خروجی ظاهر می‌شود.
2. قابل توسعه با پارامتر → می‌توان نسخه پارامتری ساخت تا پهنه‌ای بیت تغییر کند.
3. رفتار امن در حالت → جلوگیری از لاج ناخواسته و خروجی ناشناخته.

ALU (Arithmetic Logic Unit) : نام مازول

نوع واحد محاسباتی - منطقی (ترکیبی - Combinational)

The screenshot shows the ISE Project Navigator interface with the following details:

- File Menu:** File, Edit, View, Project, Source, Process, Tools, Window, Layout, Help.
- Design View:** Implementation (selected), Simulation.
- Hierarchy Tree:** xc3s50-5pq208 > Datapath (Datapath.v) > CU - ControlUnit (ControlUnit.v) > ALU - ALU (ALU.v) (highlighted).
- Code Editor:** ALU.v code listing.
- Bottom Panel:** Processes: ALU - ALU, Design Utilities, Check Syntax.

```
1 `timescale 1ns / 1ps
2
3 module ALU (
4     input [31:0] SrcA,
5     input [31:0] SrcB,
6     input [2:0] ALUControl,
7     output reg [31:0] ALUResult,
8     output logic Zero
9 );
10    always @(*) begin
11        case (ALUControl)
12            3'b000: ALUResult = SrcA & SrcB;                                // AND
13            3'b001: ALUResult = SrcA | SrcB;                                  // OR
14            3'b010: ALUResult = SrcA + SrcB;                                 // ADD
15            3'b110: ALUResult = SrcA - SrcB;                                // SUB
16            3'b111: ALUResult = (SrcA < SrcB) ? 1 : 0;                      // SLT
17            default: ALUResult = 32'hXXXXXXXX;                            // Undefined
18        endcase
19    end
20
21    assign Zero = (ALUResult == 32'b0);
22
23 endmodule
24
```

```

1 `timescale 1ns / 1ps
2
3 module ALU_tb;
4
5   reg [31:0] SrcA;
6   reg [31:0] SrcB;
7   reg [2:0] ALUControl;
8   wire [31:0] ALUResult;
9   wire Zero;
10
11   ALU uut (.SrcA(SrcA), .SrcB(SrcB), .ALUControl(ALUControl), .ALUResult(ALUResult), .Zero(Zero));
12
13   initial begin
14     SrcA = 32'd15;
15     SrcB = 32'd10;
16
17     // AND
18     ALUControl = 3'b000; #10;
19     // OR
20     ALUControl = 3'b001; #10;
21     // ADD
22     ALUControl = 3'b010; #10;
23     // SUB
24     ALUControl = 3'b110; #10;
25     // SLT
26     ALUControl = 3'b111; #10;
27
28     // Zero flag
29     SrcA = 32'd42;
30     SrcB = 32'd42;
31     ALUControl = 3'b110; // SUB ? Zero = 1
32     #10;
33
34     // (default)
35     ALUControl = 3'b011; #10;
36     $stop;

```

عملکرد کلی:

- این مازول عملیات اصلی پردازنده MIPS Multi-Cycle مانند AND ، OR، جمع، تفریق و مقایسه Set Less) (Than را انجام می‌دهد.
- بر اساس سیگنال کنترلی [2:0] ALUControl مشخص می‌شود که ALU چه عملیاتی روی دو ورودی SrcA و SrcB انجام دهد.
- سیگنال Zero زمانی فعال می‌شود که نتیجه ALU برابر صفر شود (برای دستورات شرطی مانند BEQ)

ورودی‌ها و خروجی‌ها:

• ورودی‌ها:

- ALU → عملوند اول SrcA[31:0]
- ALU → عملوند دوم SrcB[31:0]
- سیگنال کنترل برای تعیین نوع عملیات → ALUControl[2:0]

• خروجی‌ها:

- نتیجه عملیات محاسباتی/منطقی → ALUResult[31:0]
- یک بیت نشانگر صفر بودن نتیجه → Zero

: Data Path نقش در

- در مرحله Execute دستورالعمل‌ها، ALU محاسبات آدرس حافظه، نتیجه عملیات منطقی یا جمع/تفریق را انجام می‌دهد.
- سیگنال Zero برای اجرای پرش‌های شرطی (BEQ, BNE) در کنترل واحد استفاده می‌شود.

ویژگی های مهم:

1. کاملاً ترکیبی و بدون تاخیر کلاک \rightarrow خروجی سریع به روز می شود.
2. پشتیبانی از دستورات اصلی I-Type و R-Type در پردازنده MIPS.
3. قابل توسعه برای عملیات بیشتر مثل NOR، XOR و شیفت ها.
4. پرچم Zero داخلي \rightarrow ساده سازی کنترل واحد.

ماژول Mux3to1 — انتخاب ورودی برای PC

7.4 Multicycle Processor

407

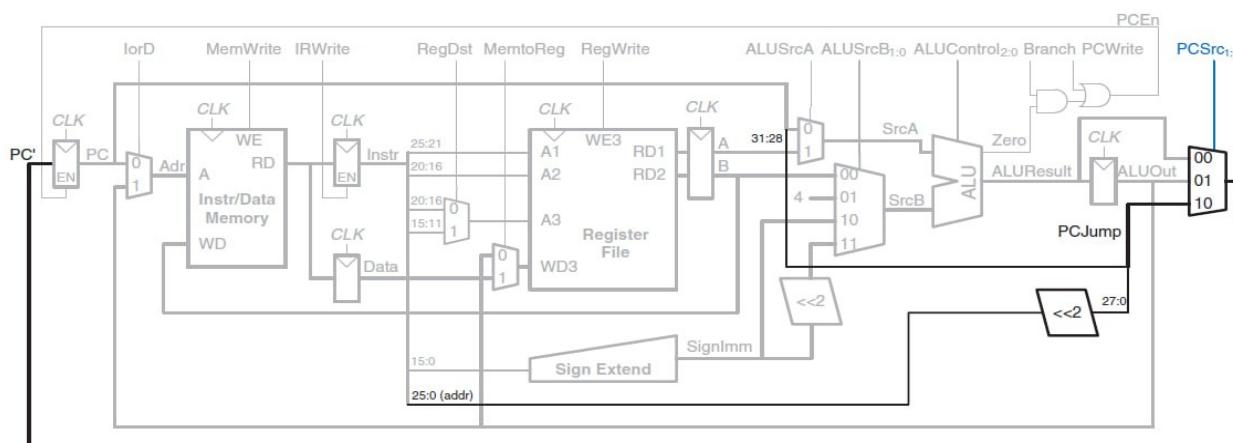


Figure 7.41 Multicycle MIPS datapath enhanced to support the *j* instruction

به *datapath* ای که داشتیم یک Mux 3to1 اضافه میکنیم تا دیتاپس بتواند دستور jump را هم پشتیبانی کند.

نقش ماژول در پردازنده:

- این مالتیپلکسر تعیین می کند مقدار بعدی PC از کجا بیاید:
- **in0 (PC+4)** \rightarrow 00: حالت عادی اجرای متوالی دستورها
- **in1 (Branch)** \rightarrow 01: وقتی شرط پرش برقرار است.
- **in2 (Jump)** \rightarrow 10: وقتی دستور J یا JAL اجرا می شود

جزئیات پیاده سازی:

- ورودی‌ها : سه عدد ۳۲ بیتی که هر کدام منبعی برای آدرس بعدی PC هستند.
- sel(سیگنال انتخاب) : ۲ بیت از کنترل واحد که مشخص می‌کند کدام ورودی انتخاب شود.
- خروجی : یک عدد ۳۲ بیتی که مستقیماً به رجیستر PC وصل می‌شود.

چرا در معماری چندسیکل MIPS نیاز داریم؟

- چون PC همیشه از یک مسیر نمی‌آید:
 - مسیر 4 برای ادامه اجرای معمولی PC+4
 - مسیر (برای دستورهای شرطی مثل Branch (BEQ
 - مسیر Jump برای پرش مستقیم مثل (J, JAL)
- اگر این مالتیپلکسر نباشد، PC فقط یک ورودی دارد و امکان پرش وجود نخواهد داشت.

Control Unit:

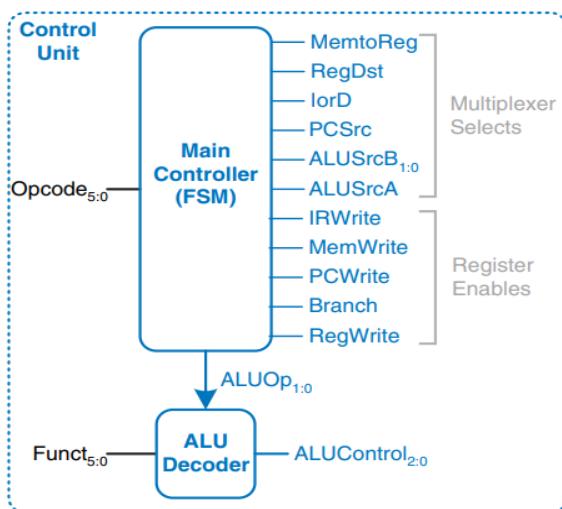
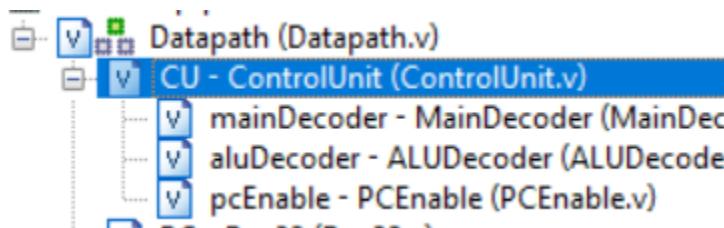


Figure 7.28 Control unit internal structure



یکی از اصلی‌ترین بخش‌های پردازنده است که وظیفه تولید و مدیریت سیگنال‌های کنترلی را بر عهده دارد. این سیگنال‌ها مسیر جریان داده (data path) و رفتار اجزای مختلف پردازنده مثل ALU ، Register File و Memory را مشخص می‌کنند. در معماری MIPS multicycle processor، اجرای هر دستور در چندین سیکل کلک انجام می‌شود (برخلاف single-cycle که همه چیز در یک سیکل اتفاق می‌افتد) . به همین دلیل، واحد کنترل باید بداند که در هر مرحله چه عملی باید صورت بگیرد.

Control Unit مغز پردازنده است. این واحد به دستور نگاه می‌کند، تشخیص می‌دهد باید چه کاری انجام شود، و سپس با فعال‌سازی یا غیرفعال‌سازی سیگنال‌ها، اجزای مختلف پردازنده را هماهنگ می‌کند.

از سه جزء اصلی تشکیل شده است:

- .1 .Op: تعیین توالی وضعیت‌ها و تولید سیگنال‌های کنترلی سطح بالا بر اساس .Op
- .2 .ALUCtrl: دیکود ALUOp و (برای R-type) Funct فیلد به سیگنال نهایی ALUOp
- .3 .PCEnable: گیت کردن نوشتمن روی PC با توجه به PCWrite و شرط Branch (Branch & Zero)

ماژول ControlUnit این سه جزء را یکپارچه می‌کند و سیگنال‌های نهایی دیتاپیث را می‌سازد.

1) MainDecoder (FSM)

```

1 `timescale 1ns / 1ps
2
3 module MainDecoder (clk, reset, Op, IorD, RegDst, PCSource, MemtoReg, ALUSrcA, ALUSrcB,
4   IRWrite, MEMWRITE, RegWrite, PCWrite, Branch, ALUOp);
5
6   input clk;
7   input reset;
8   input [5:0] Op;
9
10  // Multiplexer Selects
11  output reg IorD;
12  output reg RegDst;
13  output reg [1:0] PCSource;
14  output reg MemtoReg;
15  output reg ALUSrcA;
16  output reg [1:0] ALUSrcB;
17
18  // Register Enables
19  output reg IRWrite;
20  output reg MEMWRITE;
21  output reg RegWrite;
22  output reg PCWrite;
23  output reg Branch;
24  output reg [1:0] ALUOp;
25
26
27  //states
28  parameter FETCH = 4'b0000;
29  parameter DECODE = 4'b00001;
30  parameter MEMADR = 4'b0010;
31  parameter MEMREAD = 4'b0011;
32  parameter MEMWRITEBACK = 4'b0100;
33  parameter MEMACCESS = 4'b0101; // sw
34  parameter EXECUTE = 4'b0110;
35  parameter ALUWRITEBACK = 4'b0111; // RTYPE END
36  parameter BRANCH = 4'b1000;
37  parameter JUMPEND = 4'b1001;
38  parameter ADDIEXECUTE = 4'b1010;
39  parameter ADDIEND = 4'b1011;
40

```

```

41 // OP Code
42 parameter RTYPE = 6'b000000;
43 parameter LW = 6'b100011;
44 parameter SW = 6'b101011;
45 parameter BEQ = 6'b0000100;
46 parameter JUMP = 6'b0000010;
47 parameter ADDI = 6'b001000;
48
49 reg [3:0] state;
50 reg [3:0] nextstate;
51
52 always@(posedge clk)
53 if (reset)
54 state <= FETCH;
55 else
56 state <= nextstate;
57
58
59 always@(state or Op) begin
60 case (state)
61 FETCH: nextstate = DECODE;
62 DECODE: case(Op)
63 //OpCode
64 LW: nextstate = MEMADR;
65 SW: nextstate = MEMADR;
66 RTYPE: nextstate = EXECUTE;
67 BEQ: nextstate = BRANCH;
68 JUMP: nextstate = JUMPEND;
69 ADDI: nextstate = ADDIEXECUTE;
70 default: nextstate = FETCH;
71 endcase
72 MEMADR: case(Op)
73 LW: nextstate = MEMREAD;
74 SW: nextstate = MEMACCESS;
75 default: nextstate = FETCH;
76 endcase
77
78 MEMREAD: nextstate = MEMWRITEBACK;
79 MEMWRITEBACK: nextstate = FETCH;
80 MEMACCESS: nextstate = FETCH;
81 EXECUTE: nextstate = ALUWRITEBACK;
82 ALUWRITEBACK: nextstate = FETCH;
83 BRANCH: nextstate = FETCH;
84 JUMPEND: nextstate = FETCH;
85 ADDIEXECUTE: nextstate = ADDIEND;
86 default: nextstate = FETCH;
87 endcase
88 end
89
90 always@(state) begin
91 IorD=1'b0; MEMWRITE=1'b0; MemtoReg=1'b0; IRWrite=1'b0; PCSource=2'b00;
92 ALUSrcB=2'b00; ALUSrcA=1'b0; RegWrite=1'b0; RegDst=1'b0;
93 PCWrite=1'b0; Branch=1'b0; ALUOp=2'b00;
94
95
96 case (state)
97 FETCH:
98 begin
99 IRWrite = 1'b1;
100 ALUSrcB = 2'b01;
101 PCWrite = 1'b1;
102 end
103 DECODE:
104 ALUSrcB = 2'b11;
105 MEMADR:
106 begin
107 ALUSrcA = 1'b1;
108 ALUSrcB = 2'b10;
109 end
110 MEMREAD:
111 begin
112 IorD = 1'b1;
113 end
114
115
116

```

```

117
118     MEMWRITEBACK:
119     begin
120         RegWrite = 1'b1;
121         MemtoReg = 1'b1;
122     end
123
124     MEMACCESS:
125     begin
126         MEMWRITE = 1'b1;
127         IorD = 1'b1;
128     end
129
130     EXECUTE:
131     begin
132         ALUSrcA = 1'b1;
133         ALUOp = 2'b10;
134     end
135
136     ALUWRITEBACK:
137     begin
138         RegDst = 1'b1;
139         RegWrite = 1'b1;
140     end
141
142     BRANCH:
143     begin
144         ALUSrcA = 1'b1;
145         ALUOp = 2'b01;
146         Branch = 1'b1;
147         PCSource = 2'b01;
148     end
149
150     JUMPEND:
151     begin
152         PCWrite = 1'b1;
153         PCSource = 2'b10;
154     end
155

```

```

156     ADDIEXECUTE:
157     begin
158         ALUSrcA = 1'b1;
159         ALUSrcB = 2'b10;
160     end
161
162     ADDIEND:
163     begin
164         RegWrite = 1'b1;
165     end
166
167     endcase
168 end
169 endmodule

```

- **Inputs:**
clk, reset, Op[5:0]
- **Outputs (Datapath controls):**
lOrD, RegDst, PCSource[1:0], MemtoReg, ALUSrcA, ALUSrcB[1:0],
IRWrite, MEMWRITE, RegWrite, PCWrite, Branch, ALUOp[1:0]

State Encoding:

FETCH=0000, DECODE=0001, MEMADR=0010, MEMREAD=0011, MEMWRITEBACK=0100,
 MEMACCESS=0101, EXECUTE=0110, ALUWRITEBACK=0111, BRANCH=1000,
 JUMPEND=1001, ADDIEXECUTE=1010, ADDIEND=1011

Opcode Map:

RTYPE=000000, LW=100011, SW=101011, BEQ=000100, JUMP=000010, ADDI=001000

Transition Logic (خلاصه):

- FETCH → DECODE (همیشه)
- DECODE → MEMADR برای LW/SW
- DECODE → EXECUTE برای RTYPE
- DECODE → BRANCH برای BEQ
- DECODE → JUMPEND برای J
- DECODE → ADDIEXECUTE برای ADDI
- سایر گذارها مطابق کد، همگی به FETCH باز می‌گردند.

: Output Logic per State (Micro-ops)

نکته: در ابتدای بلوک خروجی‌ها همه‌ی سیگنال‌ها صفر می‌شوند تا latch شکل نگیرد.

: FETCH •

(IRWrite=1 (IRDستور به لود) ○
 ALU = PC + 4 ⇒ (ثابت 4) ○ ALUSrcB=01 ○

PCWrite=1, PCSOURCE=00 \Rightarrow PC \leftarrow PC+4 ○
 (آدرس حافظه از IorD=0 ○

:DECODE .

(ALUOut Branch target دار)	ALUSrcB=11 \Rightarrow ALU = PC + (SignImm << 2)	○
	برای (LW/SW) MEMADR	●
ALUSrcA=1 (A), ALUSrcB=10 (SignImm)	\Rightarrow EA = A + SignImm	○
	(مرحله خواندن) (LW) MEMREAD	●
	ALUOut \Rightarrow آدرس حافظه از IorD=1	○
	(پایان) (LW) MEMWRITEBACK	●
RegWrite=1, MemtoReg=1	\Rightarrow rt \leftarrow MDR	○
	(SW) (برای) MEMACCESS	●
IorD=1, MEMWRITE=1	\Rightarrow Mem[ALUOut] \leftarrow RegB	○
	EXECUTE (R-type)	●
ALUSrcB=00 (Funct)	ALUSrcA=1, ALUOp=10 (ضمنی)	○
	(R-type) (پایان) ALUWRITEBACK	●
RegDst=1, RegWrite=1, MemtoReg=0	\Rightarrow rd \leftarrow ALUResult	○
	BRANCH (BEQ)	●
ALUSrcA=1, ALUOp=01	\Rightarrow ALU = A - B	○
Branch target Zero=1 . PC \Rightarrow Branch=1, PCSOURCE=01	○	●
Branch target می باشد .		●

JUMPEND (J) •

PCWrite=1, PCSOURCE=10 \Rightarrow PC \leftarrow JumpAddress 
ADDIEXECUTE / ADDIEND 

ADDIEXECUTE: ALUSrcA=1, ALUSrcB=10, ALUOp=00 \Rightarrow A + SignImm o

ADDIEND: RegWrite=1, RegDst=0, MemtoReg=0 \Rightarrow rt \leftarrow ALUResult \circ

نکات طراحی:

- **Reset** در کد شما synchronous است (داخل $\@$ posedge clk)، و روی state FETCH ست می‌شود.
- مقداردهی پیشفرض خروجی‌ها در ابتدای بلوک always@(state) از ایجاد latch جلوگیری می‌کند.
- برای opcode‌های نامعتبر، ماشین حالت به FETCH پر می‌گردد.

2) ALUDecoder:

```
1 `timescale 1ns / 1ps
2
3 module ALUDecoder(ALUOp, Funct, ALUCtrl);
4
5   input [1:0] ALUOp;
6   input [5:0] Funct; //for R-type instruction
7   output reg [2:0] ALUCtrl;
8
9   always@(ALUOp or Funct) begin
10     casez({ALUOp, Funct})
11       8'b00_???????: ALUCtrl = 4'b010; // +
12       8'b01_???????: ALUCtrl = 4'b110; // -
13       8'b1?_??0000: ALUCtrl = 4'b010; // add
14       8'b1?_??0010: ALUCtrl = 4'b110; // sub
15       8'b1?_??0100: ALUCtrl = 4'b000; // and
16       8'b1?_??0101: ALUCtrl = 4'b001; // or
17       8'b1?_??1010: ALUCtrl = 4'b111; // slt
18       default: ALUCtrl = 3'b000;
19     endcase
20   end
21
22 endmodule
23
```

- **Inputs:** ALUOp[1:0], Funct[5:0]
- **Output:** ALUCtrl[2:0] (ALU: 000=AND, 001=OR, 010=ADD, 110=SUB, 111=SLT) نگاشت به

Functionality:

اگر (LW/SW/ADDI برای ALUOp=00 → ADD •
اگر (BEQ برای ALUOp=01 → SUB •
اگر نوع عملیات از :Funct (R-type) → ALUOp=1x •

- ADD (010) → 100000
- SUB (110) → 100010
- AND (000) → 100100
- OR (001) → 100101
- SLT (111) → 101010

: ALU با خوانی هم

کد ALU دقیقاً همین کدگذاری را انتظار دارد (AND=000, OR=001, ADD=010, SUB=110, SLT=111)، بنابراین خروجی این دیکودر با ALU هم راستاست.

3) PCEnable:

```
1 `timescale 1ns / 1ps
2
3 module PCEnable (
4     input PCWrite,
5     input Branch,
6     input Zero,
7     output PCEn
8 );
9     assign PCEn = PCWrite | (Branch & Zero);
10 endmodule
11
```

```
1 `timescale 1ns / 1ps
2
3 module PCEnable_tb;
4     reg PCWrite, Branch, Zero;
5     wire PCEn;
6
7     PCEnable dut (
8         .PCWrite(PCWrite),
9         .Branch(Branch),
10        .Zero(Zero),
11        .PCEn(PCEn)
12    );
13
14 initial begin
15
16     #10;PCWrite = 0; Branch = 0; Zero = 0;
17     #10;PCWrite = 1; Branch = 0; Zero = 0;
18     #10;PCWrite = 0; Branch = 1; Zero = 0;
19     #10;PCWrite = 0; Branch = 1; Zero = 1;
20     #10;PCWrite = 1; Branch = 1; Zero = 1;
21     #10;$stop;
22 end
23 endmodule
```

- **Inputs:** PCWrite, Branch, Zero
- **Output:** PCEn

• در . (JUMP ,PCWrite=1 \Rightarrow PCEn=1 و FETCH بدون شرط)

• در BRANCH، فقط اگر نتیجه ALU صفر شود (Zero=1) و Branch=1، آنگاه PC به آدرس بروزرسانی می‌شود.

• در سایر حالتهای PC لج نمی‌شود، بنابراین PCEn=0.

4) ControlUnit (Integration):

```
3 module ControlUnit(clk, reset, Op, IorD, RegDst, PCSource, MemtoReg, ALUSrcA, ALUSrcB,
4     IRWrite, MEMWRITE, RegWrite,
5     Funct, ALUCtrl,
6     Zero, PCEn);
7
8     input clk;
9     input reset;
10    input [5:0] Op;
11
12    output IorD;
13    output RegDst;
14    output [1:0] PCSource;
15    output MemtoReg;
16    output ALUSrcA;
17    output [1:0] ALUSrcB;
18
19    output IRWrite;
20    output MEMWRITE;
21    output RegWrite;
22
23    input [5:0] Funct;
24    output [2:0] ALUCtrl;
25
26    input Zero;
27    output PCEn;
28
29    wire Branch, PCWrite;
30    wire [1:0]ALUOp;
31
32 MainDecoder mainDecoder(clk, reset, Op, IorD, RegDst, PCSource, MemtoReg, ALUSrcA, ALUSrcB,
33     IRWrite, MEMWRITE, RegWrite, PCWrite, Branch, ALUOp);
34 ALUDecoder aluDecoder(ALUOp, Funct, ALUCtrl);
35 PCEnable pcEnable(PCWrite, Branch, Zero, PCEn);
36
37 endmodule
```

وظایف اصلی Control Unit

1. **Decoding دستور** : بعد از اینکه یک دستور از حافظه واکشی (fetch) شد، واحد کنترل کد عملیاتی (opcode) را بررسی می‌کند و نوع دستور (J-type, I-type, R-type) را تشخیص می‌دهد.
2. **تولید سیگنال‌های کنترلی** : بر اساس نوع دستور و فاز اجرای آن (fetch, decode, execute, memory, write-back) سیگنال‌هایی مثل (RegWrite, ALUSrc, ALUOp, MemRead, MemWrite, PCSrc, ALUOp, MemtoReg) تولید می‌کند.
3. **مدیریت مراحل اجرای دستور CU** : به صورت یک Finite State Machine (FSM) عمل می‌کند. یعنی با توجه به حالت فعلی (state) و opcode، به حالت بعدی می‌رود و سیگنال‌های مناسب را فعال می‌کند.

ساختار کلی:

- **Input**ها: شامل Opcode، بیت‌های خاصی از دستور مثل (R-type funct) و گاهی شرایط ALU (مثل Zero) می‌شوند.
- **Output**ها: سیگنال‌های کنترلی که به اجزای datapath داده می‌شوند.
- **FSM**: برای مدیریت سیکل‌های اجرای دستور.

- **Inputs:** clk, reset, Op[5:0], Funct[5:0], Zero

- **Outputs:**

Datapath controls: IorD, RegDst, PCSOURCE[1:0], MemtoReg, ALUSrcA, ALUSrcB[1:0], IRWrite, MEMWRITE, RegWrite

ALU control: ALUctrl[2:0]

PC gating: PCEn

اتصال داخلی:

- سیگنال‌های سطح بالا + ALUOp را می‌دهد.
- MainDecoder از ALUCtrl و Funct و ALUOp نهایی را تولید می‌کند.
- ALUDecoder با PCWrite، Branch، Zero و PCEnable سیگنال PC را برای رجیستر PC می‌سازد.

توالی سیکل‌ها (نمونه):

- **LW:** FETCH → DECODE → MEMADR → MEMREAD → MEMWRITEBACK → FETCH
- **SW:** FETCH → DECODE → MEMADR → MEMACCESS → FETCH
- **R-type:** FETCH → DECODE → EXECUTE → ALUWRITEBACK → FETCH
- **BEQ:** FETCH → DECODE → BRANCH → FETCH
- **J:** FETCH → DECODE → JUMPEND → FETCH
- **ADDI:** FETCH → DECODE → ADDIEXECUTE → ADDIEND → FETCH

Datapath Module:

```
1 `timescale 1ns / 1ps
2
3 module Datapath(input clk, reset, output Zero);
4
5     wire [31:0] Instr;
6     wire [5:0] Op = Instr [31:26];
7     wire [5:0] Funct = Instr [5:0];
8     wire [2:0] ALUCtrl;
9     wire [1:0] ALUSrcB;
10    wire [1:0] PCSource;
11    wire IorD, RegDst, MemtoReg, ALUSrcA, IRWrite, MemWrite, RegWrite, PCEn;
12    ControlUnit CU (clk, reset, Op, IorD, RegDst, PCSource, MemtoReg, ALUSrcA, ALUSrcB,
13        IRWrite, MemWrite, RegWrite,
14        Funct, ALUCtrl,
15        Zero, PCEn
16    );
17
18    wire [31:0] PCIn;
19    wire [31:0] PCOut;
20    Reg32 PC (clk, PCEn, PCIn, PCOut);
21
22    wire [31:0] ALUOut;
23    wire [31:0] Addr;
24    Mux2tol MUXMem (PCOut, ALUOut, IorD, Addr);
25
26    wire [31:0] RegB;
27    wire [31:0] RD;
28    Memory mem (clk, MemWrite, Addr, RegB, RD);
29
30    Reg32 IRReg (clk, IRWrite, RD, Instr);
31
32    wire [31:0] Data;
33    Reg32 MDR (clk, 1'b1, RD, Data);
34
35    wire [4:0] A1 = Instr[25:21];
36    wire [4:0] A2 = Instr[20:16];
37    wire [4:0] A3;
38    Mux2tol #(5) MuxRegDst (Instr[20:16], Instr[15:11], RegDst, A3);
39
40    wire [31:0] WD3;
41    Mux2tol MuxMemtoReg (ALUOut, Data, MemtoReg, WD3);
42
43    wire [31:0] RD1;
44    wire [31:0] RD2;
45    RegisterFile RegFile (clk, RegWrite, A1, A2, A3, WD3, RD1, RD2);
46
47    wire [31:0] RegA;
48    Reg32 RegisterA (clk, 1'b1, RD1, RegA);
49
50    Reg32 RegisterB (clk, 1'b1, RD2, RegB);
51
52    wire [31:0] SignImm;
53    SignExtend SE (Instr[15:0], SignImm);
54
55    wire [31:0] SrcA;
56    Mux2tol MuxALUSrcA (PCOut, RegA, ALUSrcA, SrcA);
57
58    wire [31:0] ShiftSignImm;
59    ShiftLeft2 SHL (SignImm, ShiftSignImm);
60
61    wire [31:0] SrcB;
62    Mux4tol MuxALUSrcB (RegB, 32'd4, SignImm, ShiftSignImm, ALUSrcB, SrcB);
63
64    wire [31:0] ALUResult;
65    ALU ALU (SrcA, SrcB, ALUCtrl, ALUResult, Zero);
66
67    Reg32 RegALUOut (clk, 1'b1, ALUResult, ALUOut);
68
69    wire [31:0] ShiftJump;
70    ShiftLeft2 SHL2 (Instr, ShiftJump);
71
72    Mux3tol MuxPCSrc (ALUResult, ALUOut, {{PCOut[31:28]}}, {ShiftJump[27:0]}}, PCSource, PCIn);
73
74 endmodule
```

ماژول **Datapath** مسئول پیاده‌سازی مسیر داده (Data Path) پردازندۀ MIPS در حالت **Multicycle** است. در این طراحی، هر دستورالعمل در چندین سیکل اجرا می‌شود و سیگنال‌های کنترلی توسط **Control Unit** تولید می‌شوند تا اجزای datapath به صورت هماهنگ عمل کنند.

این ماژول شامل اجزای اصلی مانند **Mux**, **ALU**, **Memory**, **Register File**, **Program Counter (PC)** و **Registers** است که داده‌ها را بین مراحل مختلف اجرای دستور نگه می‌دارند.

ورودی‌ها و خروجی‌ها:

• ورودی‌ها:

- کلک سیستم: clk
- سیگنال ریست: reset

• خروجی‌ها:

- خروجی Zero: Zero
- خروجی ALU: Zero (breq)

بخش‌های اصلی ماژول:

Control Unit .1

ControlUnit CU (...);

کنترل یونیت سیگنال‌های کنترلی مثل RegWrite, MemWrite, IorD, RegDst, PCSrc, ALUSrcA, ALUSrcB و همچنین ALUCtrl را تولید می‌کند.

این سیگنال‌ها رفتار datapath را در هر سیکل تعیین می‌کنند.

Program Counter (PC) .2

Reg32 PC (clk, PCEn, PCIn, PCOut);

رجیستری 32 بیتی که آدرس دستورالعمل جاری را نگه می‌دارد.

PCOut : آدرس جاری دستور را نگه می‌دارد.

PCIn : مقدار بعدی PC است که توسط MuxPCSrc انتخاب می‌شود.

PCEn : کنترل کننده فعال‌سازی PC است (با توجه به Branch یا PCWrite).

Memory.3

Mux2to1 MUXMem (PCOut, ALUOut, IorD, Adr);
Memory mem (clk, MemWrite, Adr, RegB, RD);

شامل هر دو بخش Data Memory و Instruction Memory است.

- با استفاده از مالتیپلکسر MUXMem ، آدرس حافظه یا از PCOut (برای Fetch دستور) و یا از ALUOut (برای Dستور) (Load/Store) انتخاب می شود.
- ماژول Memory داده را از حافظه می خواند (RD) یا در آن می نویسد.

Memory Data Register (MDR) و Instruction Register (IR) . 4

Reg32 IRReg (clk, IRWrite, RD, Instr);
Reg32 MDR (clk, 1'b1, RD, Data);

- دستور واکشی شده از حافظه را ذخیره می کند.
- داده خوانده شده از حافظه را نگه می دارد تا در مراحل بعدی استفاده شود (مثلاً در دستور LW).

Register File.5

RegisterFile RegFile (...);

- رجیستر فایل سه بخش دارد:
- دو خروجی خواندن (RD2 و RD1)
- یک ورودی نوشتن (کنترل شده با RegWrite)
- انتخاب رجیستر مقصد (A3) توسط RegDst انجام می شود.
- داده ای که در رجیستر نوشته می شود از طریق MemtoReg بین خروجی ALU و داده حافظه انتخاب می شود.

6. رجیستر های میانی (Register B و Register A)

Reg32 RegisterA (clk, 1'b1, RD1, RegA);
Reg32 RegisterB (clk, 1'b1, RD2, RegB);

- داده خوانده شده از رجیستر فایل در این رجیسترها ذخیره می شود تا در مراحل بعدی مثل ورودی (ALU) استفاده گردد.

Shifter و Sign Extend . 7

SignExtend SE (Instr[15:0], SignImm);
ShiftLeft2 SHL (SignImm, ShiftSignImm);

- مقدار immediate را از 16 بیت به 32 بیت گسترش می دهد.
- مقدار offset (jump branch یا offset) را دو بیت به چپ شیفت می دهد.

و انتخاب ورودی‌ها ALU. 8

Mux2to1 MuxALUSrcA (PCOut, RegA, ALUSrcA, SrcA);
Mux4to1 MuxALUSrcB (RegB, 32'd4, SignImm, ShiftSignImm, ALUSrcB, SrcB);
ALU ALU (SrcA, SrcB, ALUCtrl, ALUResult, Zero);

- ورودی اول ALU (SrcA) از بین PCOut یا RegA انتخاب می‌شود.
- ورودی دوم ALU (SrcB) می‌تواند یکی از چهار گزینه باشد:
 - RegB
 - عدد ثابت 4 (برای افزایش PC)
 - گسترش‌یافته immediate
 - شیفت‌داده شده (برای branch immediate)
- عملیات موردنظر (جمع، تفریق، and، or، slt وغیره) را انجام می‌دهد.

ALUOut Register.9

Reg32 RegALUOut (clk, 1'b1, ALUResult, ALUOut);

- خروجی ALU در این رجیستر ذخیره می‌شود تا در سیکل‌های بعدی قابل استفاده باشد مثلاً در دستو (Load/Store).

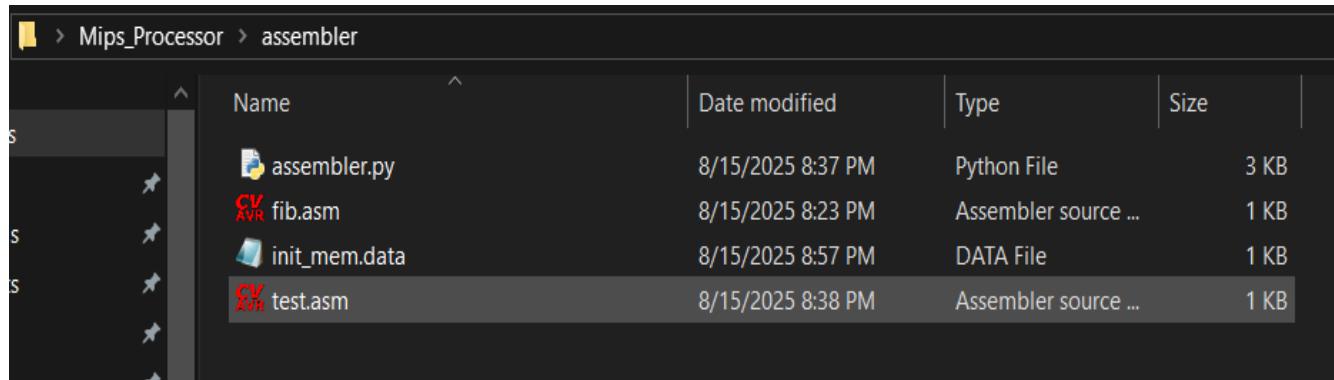
PC و انتخاب Jump Address. 10

ShiftLeft2 SHL2 (Instr, ShiftJump);
Mux3to1 MuxPCSrc (ALUResult, ALUOut, {{PCOut[31:28]}, {ShiftJump[27:0]}}, PCSrc, PCIn);

- برای پرش (Jump)، بخش 26 بیتی از دستور شیفت داده می‌شود و با بالاترین 4 بیت PC ترکیب می‌گردد.
- مالتی‌پلکسor MuxPCSrc یکی از سه گزینه را به عنوان مقدار بعدی PC انتخاب می‌کند:
 - (PC+4) برای ALUResult
 - (branch) برای ALUOut
 - Jump آدرس

- مازول MIPS در معماری چندسیکل، تمامی اجزای اصلی پردازنده را شامل می‌شود و با استفاده از سیگنال‌های کنترلی تولیدشده توسط Control Unit، وظیفه‌ی اجرای درست دستورالعمل‌ها را بر عهده دارد. ساختار مازول به گونه‌ای است که داده‌ها به وسیله رجیسترها میانی و مالتی‌پلکسراها بین اجزا حرکت کرده و مراحل Fetch، Write Back و Memory Access، Execute، Decode انجام می‌شوند.

(init_mem.data) تولید فایل Python Assembler Script



```
C: > Users > Lenovo > Desktop > Mips_Processor > assembler > assembler.py
1     asm_filename = input("Enter assembly program filename: ").strip()
2     asm = open(asm_filename, "r")
3     instruction_file = open("init_mem.data", 'w')
4
5     opcodes = {
6         "add": "0000000",
7         "sub": "0000000",
8         "and": "0000000",
9         "or": "0000000",
10        "slt": "0000000",
11        "lw": "100011",
12        "sw": "101011",
13        "beq": "000100",
14        "addi": "001000",
15        "j": "000010"
16    }
17
18    functs = {
19        "add": "1000000",
20        "sub": "100010",
21        "and": "100100",
22        "or": "100101",
23        "slt": "101010"
24    }
25
26    # Register name to number map
27    registers = {
28        "$zero": 0, "$at": 1, "$v0": 2, "$v1": 3,
29        "$a0": 4, "$a1": 5, "$a2": 6, "$a3": 7,
30        "$t0": 8, "$t1": 9, "$t2": 10, "$t3": 11,
31        "$t4": 12, "$t5": 13, "$t6": 14, "$t7": 15,
32        "$s0": 16, "$s1": 17, "$s2": 18, "$s3": 19,
33        "$s4": 20, "$s5": 21, "$s6": 22, "$s7": 23,
34        "$t8": 24, "$t9": 25, "$k0": 26, "$k1": 27,
35        "$gp": 28, "$sp": 29, "$fp": 30, "$ra": 31
36    }
```

```

38     def reg_to_5bit(reg):
39         return "{:05b}".format(registers[reg])
40
41     def int_to_16bit(value):
42         value = int(value)
43         if value < 0:
44             value = (1 << 16) + value # two's complement
45         return "{:016b}".format(value)
46
47     def int_to_26bit(value):
48         return "{:026b}".format(int(value))
49
50     instructions = asm.readlines()
51     for instr in instructions:
52         instr = instr.strip()
53         if instr == "" or instr.startswith("#"):
54             continue
55
56         instr = instr.replace(',', ' ').replace('(', ' ').replace(')', ' ').split()
57         instr[0] = instr[0].lower()
58
59         binary_instr = opcodes[instr[0]]
60
61         if instr[0] in functs: # R-type
62             binary_instr += reg_to_5bit(instr[2]) # rs
63             binary_instr += reg_to_5bit(instr[3]) # rt
64             binary_instr += reg_to_5bit(instr[1]) # rd
65             binary_instr += "00000" # shamt
66             binary_instr += functs[instr[0]] # funct
67
68         elif instr[0] in ["lw", "sw":] # I-type memory
69             binary_instr += reg_to_5bit(instr[3]) # base
70             binary_instr += reg_to_5bit(instr[1]) # rt
71             binary_instr += int_to_16bit(instr[2]) # offset
72

```

```

73         elif instr[0] in ["addi":] # I-type add immediate
74             binary_instr += reg_to_5bit(instr[2]) # rs
75             binary_instr += reg_to_5bit(instr[1]) # rt
76             binary_instr += int_to_16bit(instr[3]) # immediate
77
78         elif instr[0] == "beq": # I-type branch
79             binary_instr += reg_to_5bit(instr[1]) # rs
80             binary_instr += reg_to_5bit(instr[2]) # rt
81             binary_instr += int_to_16bit(instr[3]) # offset
82
83         elif instr[0] == "j": # J-type jump
84             binary_instr += int_to_26bit(instr[1])
85
86         instruction_file.write(binary_instr + "\n")
87         print(binary_instr)
88
89     asm.close()
90     instruction_file.close()
91

```

این اسکریپت در واقع یک **Assembler** ساده برای پردازندگی MIPS است؛ یعنی کدی که فایل اسembلی مثل

(program.asm) را می‌گیرد و معادل باینری هر دستور را به صورت خط به خط داخل یک فایل حافظه

(\$readmemb Memory) نویسد. بعد همین فایل توسط ماژول Verilog (init_mem.data) بارگذاری می‌شود.

هدف:

- دریافت کد اسembلی نوشته شده برای پردازنده MIPS
- ترجمه هر دستور به کد ماشین ۳۲ بیتی مطابق فرمت J-type, I-type, R-type
- ذخیره باینری ها در فایلی به نام init_mem.data برای بارگذاری در حافظه دستور (Instruction Memory)
- این فایل بعداً در شبیه سازی (Memory.v) خوانده می شود تا پردازنده برنامه را اجرا کند.

ساختار کلی برنامه:

1. خواندن ورودی و آماده سازی فایل خروجی

- از کاربر نام فایل اسembلی (مثلاً program.asm) گرفته می شود.
- فایل اسembلی باز می شود و خط به خط خوانده خواهد شد.
- فایل خروجی init_mem.data برای نوشتan باینری دستورها باز می شود.

2. تعریف جداول Funct و Opcode

- چون دستورات R-type همه funct = 0 دارند، ولی فرق می کند، این دو جدول جدا تعریف شده اند.
- دستورات I-type و J-type از جدول opcodes (lw, sw, addi, beq, j) استفاده می کنند.

3. تعریف جدول Register ها

- اسم رجیستر به شماره متناظر در معماری MIPS نگاشت می شود.
- مثال: \$s1 → 17 ، \$t0 → 8

4. توابع کمکی

reg_to_5bit: شماره رجیستر را به رشته ۵ بیتی تبدیل می کند.

int_to_16bit: عدد Immediate را به ۱۶ بیت (signed, two's complement) تبدیل می کند.

int_to_26bit: آدرس برای دستور ز را به ۲۶ بیت تبدیل می کند.

5. پردازش خط به خط دستورها

- دستورها یکی یکی خوانده می‌شوند.
- خطوط خالی یا Comment با (#) رد می‌شوند.
- سپس با .split() کلمات دستور جدا می‌شوند.

```
add $t2, $t0, $t1  
→ ["add", "$t2", "$t0", "$t1"]
```

6. تولید کد باینری هر نوع دستور

7. نوشتگی خروجی

- هر دستور ۳۲ بیتی به صورت یک خط در init_mem.data نوشته می‌شود.
- همزمان در Console هم چاپ می‌شود (برای Debug)

(init_mem.data) نمونه

```
PS C:\Users\Lenovo\Desktop\Mips_Processor\assembler> python .\assembler.py  
Enter assembly program filename: test.asm  
001000000001000000000000000101  
0010000000010010000000000001010  
000000010000100101010000010000  
0000000100101000010110000100010  
000000010000100101100000100100  
0000000100001001011010000100101  
00000001000010010110100000100101  
00000001000010010111000000101010  
10101100000010100000000000000000  
10001100000011110000000000000000  
000100011110101000000000000010  
0010000111101111000000000000001  
0010000000010000000000000000000000  
0000100000000000000000000000000000  
0010000000001000000000000000000000  
0010000000000000000000000000000000  
PS C:\Users\Lenovo\Desktop\Mips_Processor\assembler>
```

این فایل نشان‌دهنده‌ی همان دستورات اسمبلي به صورت باينري است:

	Address	Instruction	Binary
0	addi \$t0, \$zero, 5	001000 00000 01000 0000000000000000101	
1	addi \$t1, \$zero, 10	001000 00000 01001 00000000000000001010	
2	add \$t2, \$t0, \$t1	000000 01000 01001 01010 00000 100000	
3	sub \$t3, \$t1, \$t0	000000 01001 01000 01011 00000 100010	
4	and \$t4, \$t0, \$t1	000000 01000 01001 01100 00000 100100	
5	or \$t5, \$t0, \$t1	000000 01000 01001 01101 00000 100101	
6	slt \$t6, \$t0, \$t1	000000 01000 01001 01110 00000 101010	
7	sw \$t2, 0(\$zero)	101011 00000 01010 00000000000000000000	
8	lw \$t7, 0(\$zero)	100011 00000 01111 00000000000000000000	
9	beq \$t7, \$t2, 2	000100 01111 01010 000000000000000010	
10	addi \$t7, \$t7, 1	001000 01111 01111 00000000000000000001	
11	addi \$t0, \$zero, 100	001000 00000 01000 0000000001100100	
12	j 14	000010 000000000000000000000000000000001110	
13	addi \$t0, \$zero, 200	001000 00000 01000 0000000011001000	
14	addi \$t1, \$zero, 50	001000 00000 01001 0000000000000000110010	

این فایل ورودی حافظه (Instruction Memory) است. هر خط دقیقاً معادل یک دستور ۳۲ بیتی است. در مازول حافظه \$readmemb("init_mem.data", mem)، با دستور Verilog این فایل لود می‌شود.

Datapath_tb (Testbench)

```
1 `timescale 1ns / 1ps
2
3 module Datapath_tb;
4
5     // Inputs
6     reg clk;
7     reg reset;
8
9     // Outputs
10    wire Zero;
11
12    Datapath dut (.clk(clk),.reset(reset),.Zero(Zero));
13
14    // Clock gen
15    initial begin
16        clk = 0;
17        reset = 0;
18        #940;
19        $stop;
20    end
21    always #5 clk = ~clk;
22
23    // ---- ALU Logging ----
24    always @(posedge clk) begin
25        if (dut.ALUCtrl !== 3'bxxx) begin
26            $display("[%0t] ALU: SrcA=%08h SrcB=%08h ALU_Control=%0b -> Result=%08h",
27                      $time,
28                      dut.SrcA,
29                      dut.SrcB,
30                      dut.ALUCtrl,
31                      dut.ALUResult);
32        end
33    end
34
35    // ---- PC Logging ----
36    always @(posedge dut.IRWrite) begin
37        $display("[%0t] PC=%08h",
38                  $time,
39                  dut.PC.q);
40    end
41
42    // ---- Instruction Logging ----
43    reg [31:0] prev_inst;
44    always @(posedge clk) begin
45        if (dut.IRReg.q !== prev_inst) begin
46            prev_inst <= dut.IRReg.q;
47            $display("[%0t] Instruction=%08h",
48                      $time,
49                      dut.IRReg.q);
50        end
51    end
52
53    // ---- Register File Write Logging ----
54    always @(posedge clk) begin
55        if (dut.RegWrite && dut.RegFile.WriteReg != 5'd0)
56            $display("[%0t]      >> $%0d <= %08h",
57                      $time,
58                      dut.RegFile.WriteReg,
59                      dut.RegFile.WriteData);
60    end
61
62    // ---- Memory Access Logging ----
63    always @(posedge dut.MemWrite) begin
64        $display("[%0t]      >> MEM[%02h] <= %08h",
65                      $time,
66                      dut.mem.addr[9:2],
67                      dut.mem.write_data);
68    end
69 endmodule
```

این تستبنج اجرای یک برنامه‌ی کامل روی مازول Datapath را شبیه‌سازی می‌کند و با **probe** کردن سیگنال‌های داخلی (hierarchical references) لاغهای دقیقی از رفتار پردازنده چاپ می‌کند:

- تکامل PC و واکشی دستور (IR)
- ورودی/خروجی و کنترل ALU
- نوشتمن در Register File
- عملیات نوشتمن در حافظه (Store)

(assembler/init_mem.data) preload با فایل اولیه مثل Memory مازول در برنامه حافظه‌ی می‌کند. این تستبنج فرض می‌کند حافظه‌ی برنامه در مازول با پورت اولیه (init_mem.data) شده است.

Probe کردن سیگنال‌های داخلی:

این تستبنج از مسیرهای سلسله‌مراتبی استفاده می‌کند مانند (dut.ALUCtrl, dut.IRReg.q, dut.mem.addr) تا سیگنال‌ها/پورت‌های داخلی زیرماژول‌ها را مشاهده کند.

نمونه‌ها :

- dut.PC.q: خروجی رजیستر PC (نمونه‌ی Reg32 با پورت q)
- dut.IRReg.q: مقدار Instruction Register
- dut.RegFile.WriteReg/WriteData: پورت‌های ورودی زیرماژول RegisterFile
- dut.mem.write_data و dut.mem.addr: پورت‌های زیرماژول Memory
- dut.SrcA, dut.SrcB, dut.ALUResult, dut.ALUCtrl: وايرهای داخلی datapath

بلکهای لاغ‌گیری:

ALU Logging(1)

در هر لبه‌ی بالاروندهی کلک، اگر ALUCtrl مقدار معتبر (غیر از X) داشته باشد، ورودی‌ها/خروجی ALU لاغ می‌شوند.

استفاده از (`!==`) **case-inequality** با 3'bxxx تضمین می‌کند فقط وقتی کنترل ALU مشخص است، چاپ انجام شود.

: IRWrite در لحظه‌ی PC Logging(2

- با بالا رفتن IRWrite (معمولًا در FETCH state) مقدار فعلی PC چاپ می‌شود.
- از آن جا که IRWrite در ورود به FETCH یک پالس می‌گیرد، این چاپ معمولًا PC پس از بهروزرسانی (PC+4) را نشان می‌دهد.

: با تشخیص تغییر: Instruction Logging(3

- هر زمان مقدار IR عوض شود (دستور جدید fetch شود) ، یک خط لاغ چاپ می‌شود.
- رجیستر prev_inst جلوی چاپ‌های تکراری را می‌گیرد.

: Register File Write Logging(4

- در هر سیکل، اگر $\text{RegWrite} = 1$ و مقصد صفر نباشد، عملیات Write-Back گزارش می‌شود:
 - شماره رجیستر مقصد از پورت WriteReg
 - داده‌ی نوشته شده از پورت WriteData

- این چاپ‌ها نشان می‌دهد که کدام دستور چه مقداری را در کدام رجیستر نوشته است (برای lw, addi, R-type ...)

: Memory Write Logging(5

- با بالا رفتن MemWrite ، یک عمل Store گزارش می‌شود.
- ایندکس dut.mem.addr[9:2] word-aligned حافظه (عمق 256) را نشان می‌دهد.
- محتوایی است که در آن آدرس نوشته می‌شود.

خروجی رفتار پردازنده‌ی MIPS (Multicycle)

[5000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=0 -> Result=00000000
[5000] Instruction=00000000
[5000] PC=00000000
[15000] ALU: SrcA=00000000 SrcB=00000004 ALU_Control=10 -> Result=00000004
[25000] ALU: SrcA=00000004 SrcB=00000014 ALU_Control=10 -> Result=00000018
[25000] Instruction=20080005
[35000] ALU: SrcA=00000000 SrcB=00000005 ALU_Control=10 -> Result=00000005
[45000] ALU: SrcA=00000004 SrcB=00000000 ALU_Control=10 -> Result=00000004
[45000] >> \$8 <= 00000005
[45000] PC=00000004
[55000] ALU: SrcA=00000004 SrcB=00000004 ALU_Control=10 -> Result=00000008
[65000] ALU: SrcA=00000008 SrcB=00000028 ALU_Control=10 -> Result=00000030
[65000] Instruction=2009000a
[75000] ALU: SrcA=00000000 SrcB=0000000a ALU_Control=10 -> Result=0000000a
[85000] ALU: SrcA=00000008 SrcB=00000000 ALU_Control=10 -> Result=00000008
[85000] >> \$9 <= 0000000a
[85000] PC=00000008
[95000] ALU: SrcA=00000008 SrcB=00000004 ALU_Control=10 -> Result=0000000c
[105000] ALU: SrcA=0000000c SrcB=00014080 ALU_Control=10 -> Result=0001408c
[105000] Instruction=01095020
[115000] ALU: SrcA=00000005 SrcB=0000000a ALU_Control=10 -> Result=0000000f
[125000] ALU: SrcA=0000000c SrcB=0000000a ALU_Control=10 -> Result=00000016
[125000] >> \$10 <= 0000000f
[125000] PC=0000000c
[135000] ALU: SrcA=0000000c SrcB=00000004 ALU_Control=10 -> Result=00000010
[145000] ALU: SrcA=00000010 SrcB=00016088 ALU_Control=10 -> Result=00016098
[145000] Instruction=01285822
[155000] ALU: SrcA=0000000a SrcB=00000005 ALU_Control=110 -> Result=00000005
[165000] ALU: SrcA=00000010 SrcB=00000005 ALU_Control=10 -> Result=00000015
[165000] >> \$11 <= 00000005
[165000] PC=00000010
[175000] ALU: SrcA=00000010 SrcB=00000004 ALU_Control=10 -> Result=00000014
[185000] ALU: SrcA=00000014 SrcB=00018090 ALU_Control=10 -> Result=000180a4
[185000] Instruction=01096024
[195000] ALU: SrcA=00000005 SrcB=0000000a ALU_Control=0 -> Result=00000000
[205000] ALU: SrcA=00000014 SrcB=0000000a ALU_Control=10 -> Result=0000001e
[205000] >> \$12 <= 00000000
[205000] PC=00000014
[215000] ALU: SrcA=00000014 SrcB=00000004 ALU_Control=10 -> Result=00000018
[225000] ALU: SrcA=00000018 SrcB=0001a094 ALU_Control=10 -> Result=0001a0ac
[225000] Instruction=01096825
[235000] ALU: SrcA=00000005 SrcB=0000000a ALU_Control=1 -> Result=0000000f
[245000] ALU: SrcA=00000018 SrcB=0000000a ALU_Control=10 -> Result=00000022
[245000] >> \$13 <= 0000000f
[245000] PC=00000018
[255000] ALU: SrcA=00000018 SrcB=00000004 ALU_Control=10 -> Result=0000001c
[265000] ALU: SrcA=0000001c SrcB=0001c0a8 ALU_Control=10 -> Result=0001c0c4
[265000] Instruction=0109702a

[275000] ALU: SrcA=00000005 SrcB=0000000a ALU_Control=111 -> Result=00000001
[285000] ALU: SrcA=0000001c SrcB=0000000a ALU_Control=10 -> Result=00000026
[285000] >> \$14 <= 00000001
[285000] PC=0000001c
[295000] ALU: SrcA=0000001c SrcB=00000004 ALU_Control=10 -> Result=00000020
[305000] ALU: SrcA=00000020 SrcB=00000000 ALU_Control=10 -> Result=00000020
[305000] Instruction=ac0a0000
[315000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[315000] >> MEM[00] <= 0000000f
[325000] ALU: SrcA=00000020 SrcB=0000000f ALU_Control=10 -> Result=0000002f
[325000] PC=00000020
[335000] ALU: SrcA=00000020 SrcB=00000004 ALU_Control=10 -> Result=00000024
[345000] ALU: SrcA=00000024 SrcB=00000000 ALU_Control=10 -> Result=00000024
[345000] Instruction=8c0f0000
[355000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[365000] ALU: SrcA=00000024 SrcB=00000000 ALU_Control=10 -> Result=00000024
[375000] ALU: SrcA=00000024 SrcB=00000000 ALU_Control=10 -> Result=00000024
[375000] >> \$15 <= 0000000f
[375000] PC=00000024
[385000] ALU: SrcA=00000024 SrcB=00000004 ALU_Control=10 -> Result=00000028
[395000] ALU: SrcA=00000028 SrcB=00000008 ALU_Control=10 -> Result=00000030
[395000] Instruction=11ea0002
[405000] ALU: SrcA=0000000f SrcB=0000000f ALU_Control=110 -> Result=00000000
[405000] PC=00000030
[415000] ALU: SrcA=00000030 SrcB=00000004 ALU_Control=10 -> Result=00000034
[425000] ALU: SrcA=00000034 SrcB=00000038 ALU_Control=10 -> Result=0000006c
[425000] Instruction=0800000e
[435000] ALU: SrcA=00000034 SrcB=00000000 ALU_Control=10 -> Result=00000034
[435000] PC=00000038
[445000] ALU: SrcA=00000038 SrcB=00000004 ALU_Control=10 -> Result=0000003c
[455000] ALU: SrcA=0000003c SrcB=000000c8 ALU_Control=10 -> Result=00000104
[455000] Instruction=20090032
[465000] ALU: SrcA=00000000 SrcB=00000032 ALU_Control=10 -> Result=00000032
[475000] ALU: SrcA=0000003c SrcB=0000000a ALU_Control=10 -> Result=00000046
[475000] >> \$9 <= 00000032
[475000] PC=0000003c
[485000] ALU: SrcA=0000003c SrcB=00000004 ALU_Control=10 -> Result=00000040
[495000] ALU: SrcA=00000040 SrcB=00000000 ALU_Control=10 -> Result=00000040
[495000] Instruction=00000000
[505000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[515000] ALU: SrcA=00000040 SrcB=00000000 ALU_Control=10 -> Result=00000040
[515000] PC=00000040
[525000] ALU: SrcA=00000040 SrcB=00000004 ALU_Control=10 -> Result=00000044
[535000] ALU: SrcA=00000044 SrcB=00000000 ALU_Control=10 -> Result=00000044
[545000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[555000] ALU: SrcA=00000044 SrcB=00000000 ALU_Control=10 -> Result=00000044
[555000] PC=00000044
[565000] ALU: SrcA=00000044 SrcB=00000004 ALU_Control=10 -> Result=00000048
[575000] ALU: SrcA=00000048 SrcB=00000000 ALU_Control=10 -> Result=00000048
[585000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[595000] ALU: SrcA=00000048 SrcB=00000000 ALU_Control=10 -> Result=00000048
[595000] PC=00000048
[605000] ALU: SrcA=00000048 SrcB=00000004 ALU_Control=10 -> Result=0000004c

[615000] ALU: SrcA=0000004c SrcB=00000000 ALU_Control=10 -> Result=0000004c
[625000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[635000] ALU: SrcA=0000004c SrcB=00000000 ALU_Control=10 -> Result=0000004c
[635000] PC=0000004c
[645000] ALU: SrcA=0000004c SrcB=00000004 ALU_Control=10 -> Result=00000050
[655000] ALU: SrcA=00000050 SrcB=00000000 ALU_Control=10 -> Result=00000050
[665000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[675000] ALU: SrcA=00000050 SrcB=00000000 ALU_Control=10 -> Result=00000050
[675000] PC=00000050
[685000] ALU: SrcA=00000050 SrcB=00000004 ALU_Control=10 -> Result=00000054
[695000] ALU: SrcA=00000054 SrcB=00000000 ALU_Control=10 -> Result=00000054
[705000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[715000] ALU: SrcA=00000054 SrcB=00000000 ALU_Control=10 -> Result=00000054
[715000] PC=00000054
[725000] ALU: SrcA=00000054 SrcB=00000004 ALU_Control=10 -> Result=00000058
[735000] ALU: SrcA=00000058 SrcB=00000000 ALU_Control=10 -> Result=00000058
[745000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[755000] ALU: SrcA=00000058 SrcB=00000000 ALU_Control=10 -> Result=00000058
[755000] PC=00000058
[765000] ALU: SrcA=00000058 SrcB=00000004 ALU_Control=10 -> Result=0000005c
[775000] ALU: SrcA=0000005c SrcB=00000000 ALU_Control=10 -> Result=0000005c
[785000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[795000] ALU: SrcA=0000005c SrcB=00000000 ALU_Control=10 -> Result=0000005c
[795000] PC=0000005c
[805000] ALU: SrcA=0000005c SrcB=00000004 ALU_Control=10 -> Result=00000060
[815000] ALU: SrcA=00000060 SrcB=00000000 ALU_Control=10 -> Result=00000060
[825000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[835000] ALU: SrcA=00000060 SrcB=00000000 ALU_Control=10 -> Result=00000060
[835000] PC=00000060
[845000] ALU: SrcA=00000060 SrcB=00000004 ALU_Control=10 -> Result=00000064
[855000] ALU: SrcA=00000064 SrcB=00000000 ALU_Control=10 -> Result=00000064
[865000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[875000] ALU: SrcA=00000064 SrcB=00000000 ALU_Control=10 -> Result=00000064
[875000] PC=00000064
[885000] ALU: SrcA=00000064 SrcB=00000004 ALU_Control=10 -> Result=00000068
[895000] ALU: SrcA=00000068 SrcB=00000000 ALU_Control=10 -> Result=00000068
[905000] ALU: SrcA=00000000 SrcB=00000000 ALU_Control=10 -> Result=00000000
[915000] ALU: SrcA=00000068 SrcB=00000000 ALU_Control=10 -> Result=00000068
[915000] PC=00000068
[925000] ALU: SrcA=00000068 SrcB=00000004 ALU_Control=10 -> Result=0000006c
[935000] ALU: SrcA=0000006c SrcB=00000000 ALU_Control=10 -> Result=0000006c

Stopped at time : 940 ns

init_mem.data → Assembly readable

```
addi $t0, $zero, 1    # $t0 = 1
addi $t1, $zero, 1    # $t1 = 1
addi $t3, $zero, 2    # $t3 = 2
addi $t4, $zero, 6    # $t4 = 6
add $t2, $t0, $t1     # $t2 = $t0 + $t1
addi $t3, $t3, 1      # $t3 = $t3 + 1
beq $t3, $t4, 3       # if ($t3 == $t4) skip 3 instructions
addi $t0, $t1, 0       # $t0 = $t1
addi $t1, $t2, 0       # $t1 = $t2
j 4                   # jump to instruction at address 4
```

1. واکشی و شروع اجرا

- در سیکل‌های ابتدایی مقدار اولیهی PC برابر صفر است و ALU محاسبهی $PC + 4$ را انجام می‌دهد.
- اولین دستور خواندهشده 20080005 است که معادل دستور addi \$t0, \$zero, 5 می‌باشد.

2. اجرای دستور ADDI

- ALU مقدار 0 (محتوای \$zero) را با عدد ثابت 5 جمع می‌کند.
- در مرحلهی WB مقدار 5 در ثبات \$t0 (یعنی \$8) ذخیره می‌شود.

3. اجرای دستور دوم (ADDI)

- دستور 2009000a معادل addi \$t1, \$zero, 10 واکشی می‌شود.
- در پایان، مقدار 10 در ثبات \$t1 (یعنی \$9) نوشته می‌شود.

4. اجرای دستور R-type (ADD)

- دستور 01095020 معادل add \$t2, \$t0, \$t1 است.
- عمل جمع $10 + 5$ را انجام داده و نتیجهی 15 در ثبات \$t2 ذخیره می‌شود.

5. اجرای دستور R-type (SUB)

- دستور 01285822 معادل sub \$t3, \$t1, \$t0 است.
- حاصل $10 - 5 = 5$ در ثبات \$t3 ذخیره می‌شود.

6. اجرای دستور AND

- دستور 01096024 معادل and \$t4, \$t0, \$t1 است.

- نتیجه‌ی 10 AND 5 برابر صفر است و در \$t4 ذخیره می‌شود.

7. اجرای دستور OR

- دستور 01096825 معادل \$t5, \$t0, \$t1 or \$t5 است.
- حاصل \$t5 = 5 در OR 15 نوشته می‌شود.

8. اجرای دستور SLT

- دستور 0109702a معادل slt \$t6, \$t0, \$t1 است.
- چون $10 < 5$ ، مقدار 1 در ثبات \$t6 قرار می‌گیرد.

9. اجرای دستور SW

- دستور ac0a0000 معادل sw \$t2, 0(\$zero) است.
- مقدار 15 (محتوای \$t2) در آدرس حافظه 0 ذخیره می‌شود.

10. اجرای دستور LW

- دستور 8c0f0000 معادل lw \$t7, 0(\$zero) است.
- مقدار 15 از حافظه خوانده شده و در ثبات \$t7 قرار می‌گیرد.

11. اجرای دستور شرطی BEQ

- دستور 11ea0002 معادل beq \$t7, \$t2, offset است.
- چون مقدار \$t7 و \$t2 هر دو برابر 15 هستند، شرط برقرار شده و پرش انجام می‌شود PC به آدرس مقصد تغییر می‌کند.

12. اجرای دستور JUMP

- دستور 0800000e معادل **j 0x00000038** است.
- به آدرس 0x38 جهش داده می‌شود و اجرای برنامه از آن نقطه ادامه پیدا می‌کند.

13. پایان برنامه

- در نهایت چند دستور NOP (00000000) اجرا می‌شود و پردازنده در سیکل 940 ns متوقف می‌گردد.

جمع‌بندی

خروجی شبیه‌سازی نشان داد که پردازنده MIPS چندچرخه طراحی شده قادر به اجرای صحیح انواع دستورها شامل:

- دستورهای Immediate مثل (ADDI)
- دستورهای R-type مثل (SLT, OR, AND, SUB, ADD)
- دستورهای حافظه‌ای (SW, LW)
- دستورهای کنترلی (JUMP, BEQ)

می‌باشد. همچنین ثبت تغییرات ثبات‌ها و حافظه در لگ سیمولیشن تأییدکنندهٔ صحت عملکرد مأذول‌های کنترل، ALU، رجیستر فایل و حافظه است.

