杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

控制器

执行 MIPS 指令的处理器

单周期数据通路和控制器

CPU设计

执行 MIPS 指令的处理器：单周期数据通路和控制器

单周期 MIPS CPU （即每个周期执行一条指令）

- 主要功能部件及数据通路
  - 取指令（实验七）
  - R 型指令
  - I 型立即数寻址指令
  - I 型访存指令
  - I 型条件转移指令
  - J 型无条件转移指令
  - J 型子程序调用和返回指令

计算机体系结构

- 冯·诺依曼体系结构的计算机，将程序指令和数据存储在同一个存储器，指令和数据的宽度相同，指令计数器和数据地址指针指向同一个存储器的不同物理位置，这种结构又称为普林斯顿结构。
- 采用普林斯顿结构的处理器有英特尔公司的微处理器、 ARM 公司的 ARM7 、 MIPS 公司的 MIPS 处理器等。
- 哈佛结构则是一种将程序的指令和数据分别存储在独立的存储器中的体系结构，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问。显然，哈佛结构的 CPU 可以同时读取指令和数据；大大提高了数据吞吐率，缺点是结构复杂。
- 采用哈弗结构的处理器有 Microchip 公司的 PIC 系列芯片、摩托罗拉公司的 MC68 系列、 Zilog 公司的 Z8 系列、 ATMEL 公司的 AVR 系列和安谋公司的 ARM9 、 ARM10 和 ARM11 等。
- 采用哈佛结构可以不设单独的指令寄存器

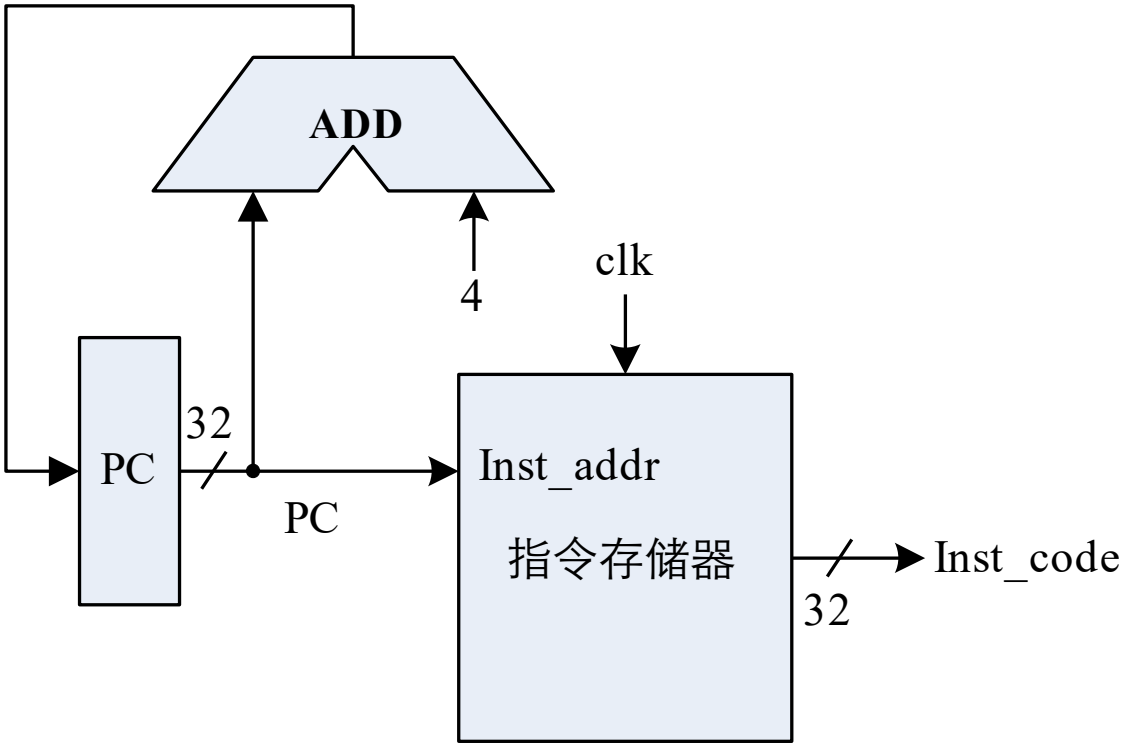
取指令的功能部件及数据通路

PC： 程序计数器，存放当前指令的地址，在每个

加法器：完成 **PC+4**，指向下一条指令

指令存储器：存放指令的机器代码，只读存储器 **ROM**

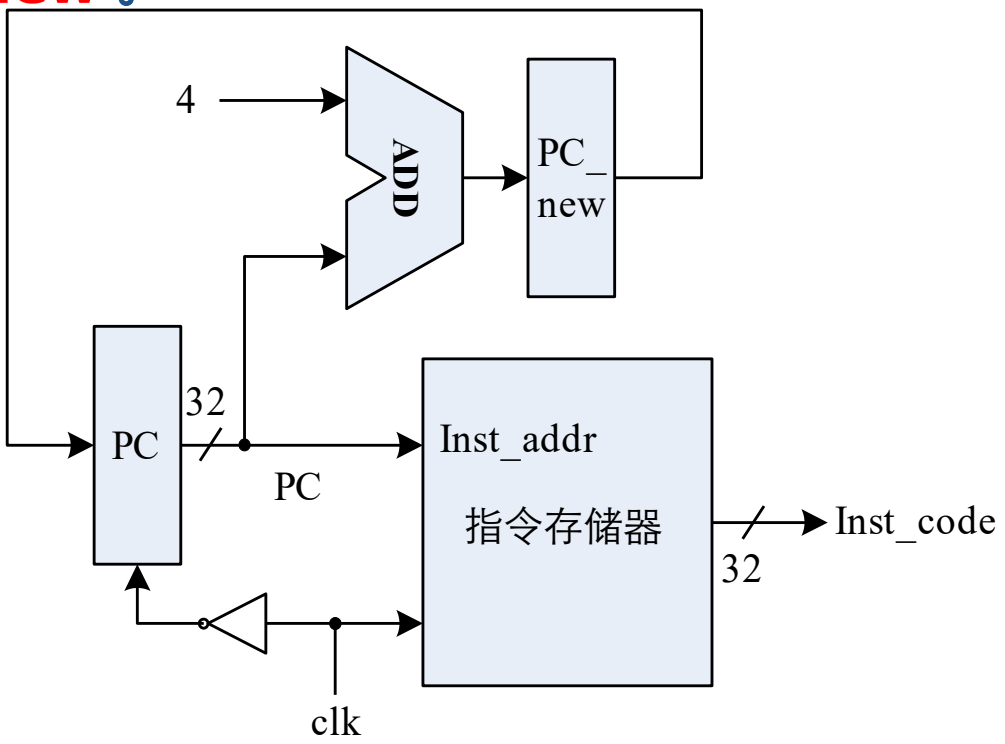
取指令操作：根据 **PC** 内容到指令存储器中取出指令，然后 **PC+4→PC**。



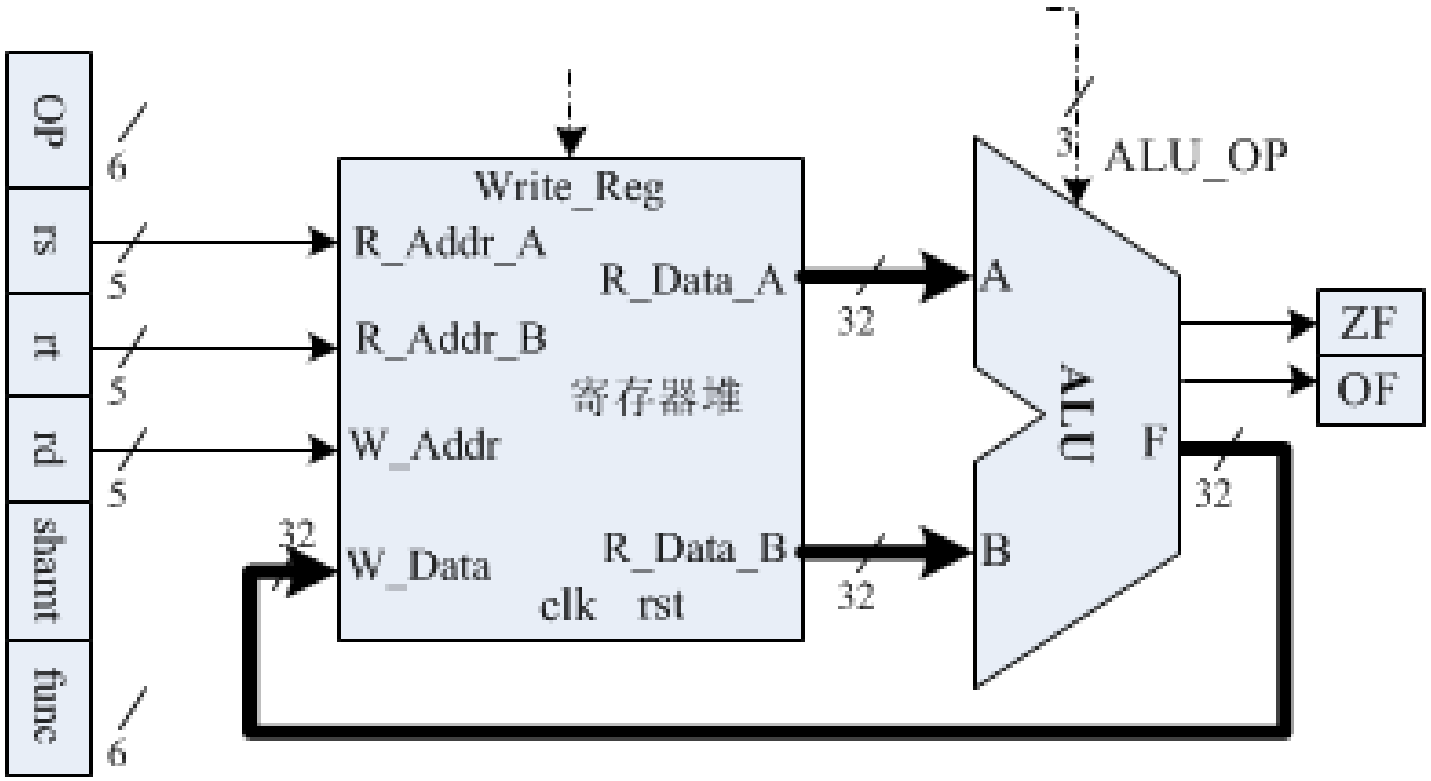
取指令的功能部件及数据通路

单周期 MIPS CPU （即每个周期执行一条指令）

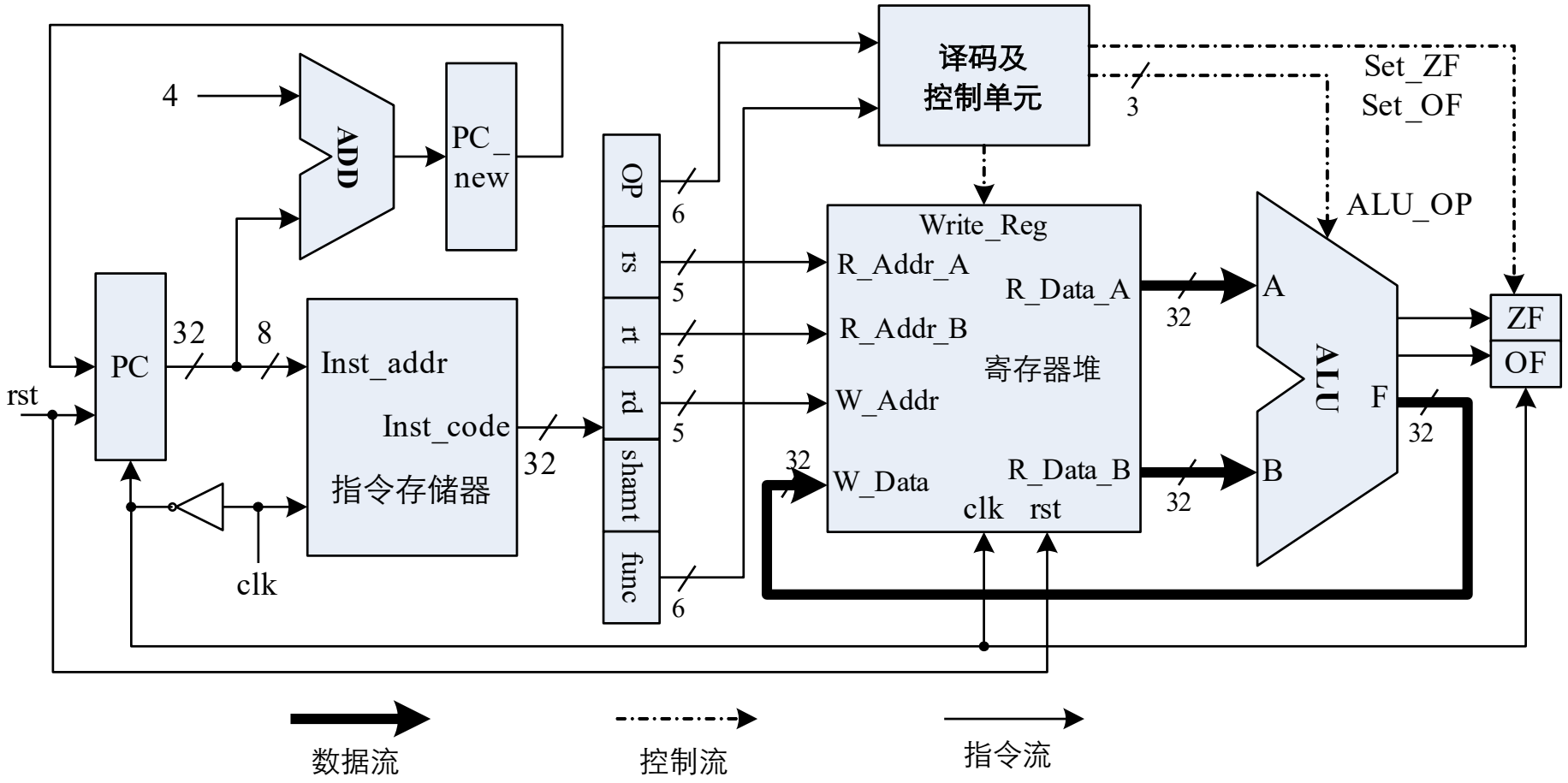
要保持在执行指令期间，指令存储器取出的指令不变，那么也就要求 PC 的内容不变，因此 PC+4 的值必须在下个指令周期开始时赋值给 PC。为此，需要设计一个暂存 PC 自增值（PC+4）的寄存器 PC_new。



执行 R 型指令的功能部件和数据通路



实现 R 型指令的 CPU 系统结构



控制器产生控制信号： **ALU_OP, Set_ZF,Set_OF, Write_Reg**

R 指令执行过程分析

- add $t1,$t2,$t3 （单周期执行过程）
  - 取指令：根据 PC 值，在 clk 上跳沿，从指令存储器中取地址为 PC 的指令 Inst_Code,PC_NEW=PC+4
  - 取操作数：从 MIPS_REGS 中取出 $t2 和 $t3 的值到 ALU 的 A 和 B
  - 执行指令：完成 F=A+B，标志位 OF 和 ZF(ALU_OP)
  - 送结果：在 clk 下跳沿，将结果 F 送到从 MIPS_REGS 的 $t1(Write_Reg=1)，更新标志位 ZF(Set_ZF=1)，更新标志位 OF(Set_OF=1)，更新 PC<=PC_NEW
- 如果 rst=1，复位信号有效，则 PC 清 0，所有寄存器清 0，ZF 和 OF 清 0

I 型立即数指令

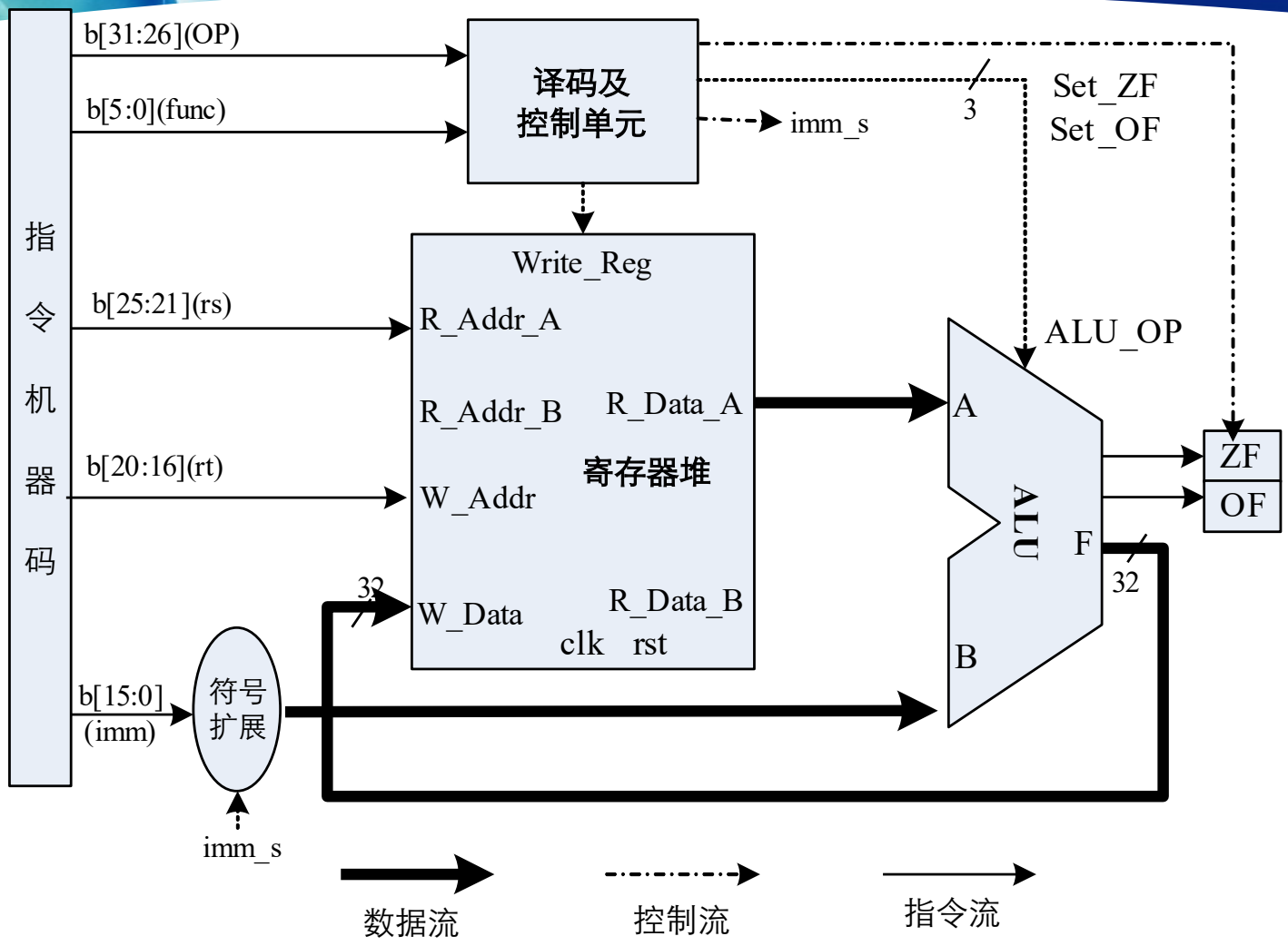
MIPS I 型立即数寻址指令格式及编码

字段	OP	rs	rt	imm	功能描述
位数	6	5	5	16	
汇编助记符	编码				
addi rt, rs, imm	001000	rs	rt	imm	算术加：rs+imm→rt
andi rt, rs, imm	001100	rs	rt	imm	逻辑与：rs&imm→rt
xori rt, rs, imm	001110	rs	rt	imm	逻辑异或：rs ⊕ imm→rt
sltiu rt, rs, imm	001011	rs	rt	imm	无符号数小于则置位：if (rs < imm) rt=1 else rt=0

imm 符号扩展

imm 0 扩展

执行 I 型立即数指令的功能部件和数据通路

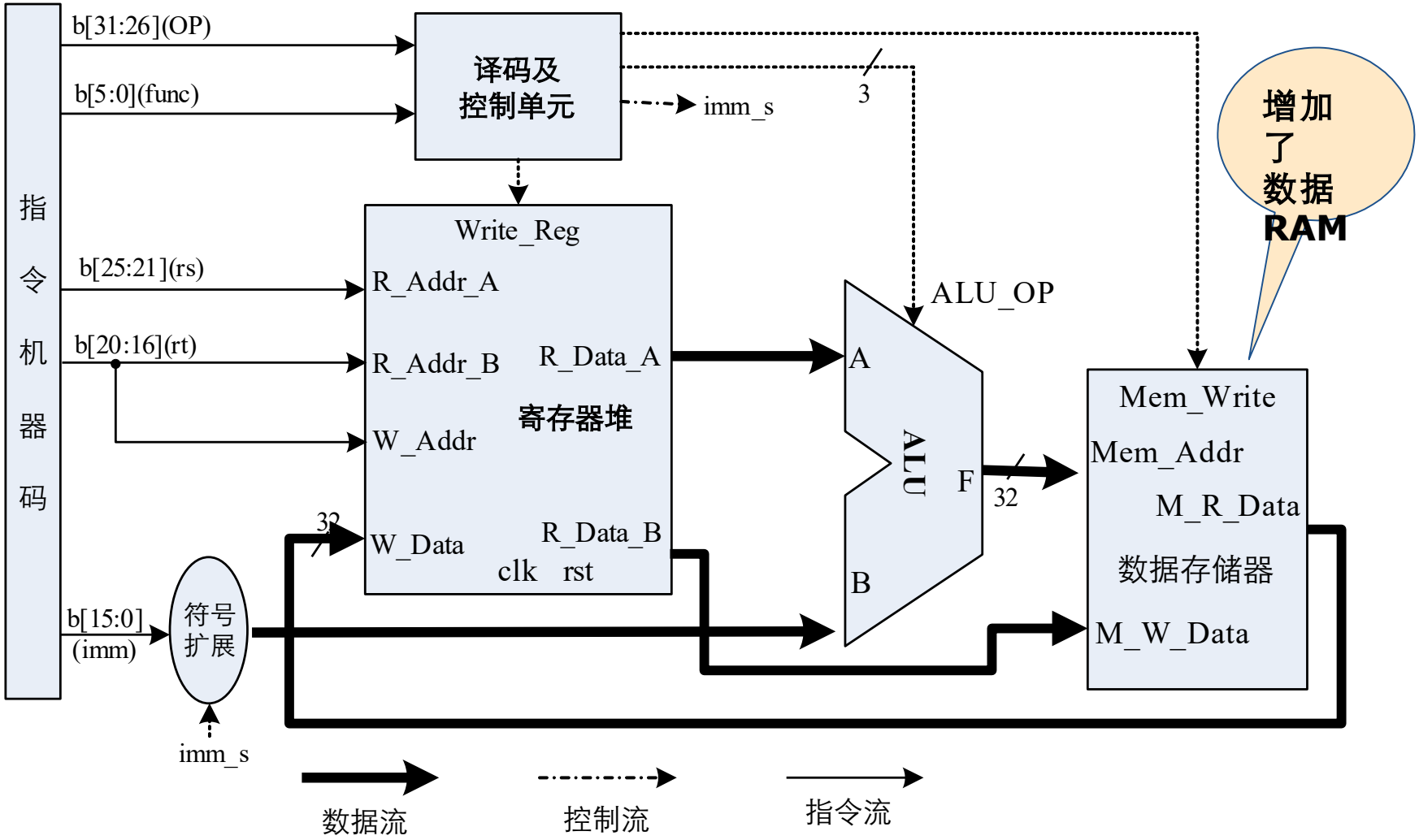


I 型访存指令

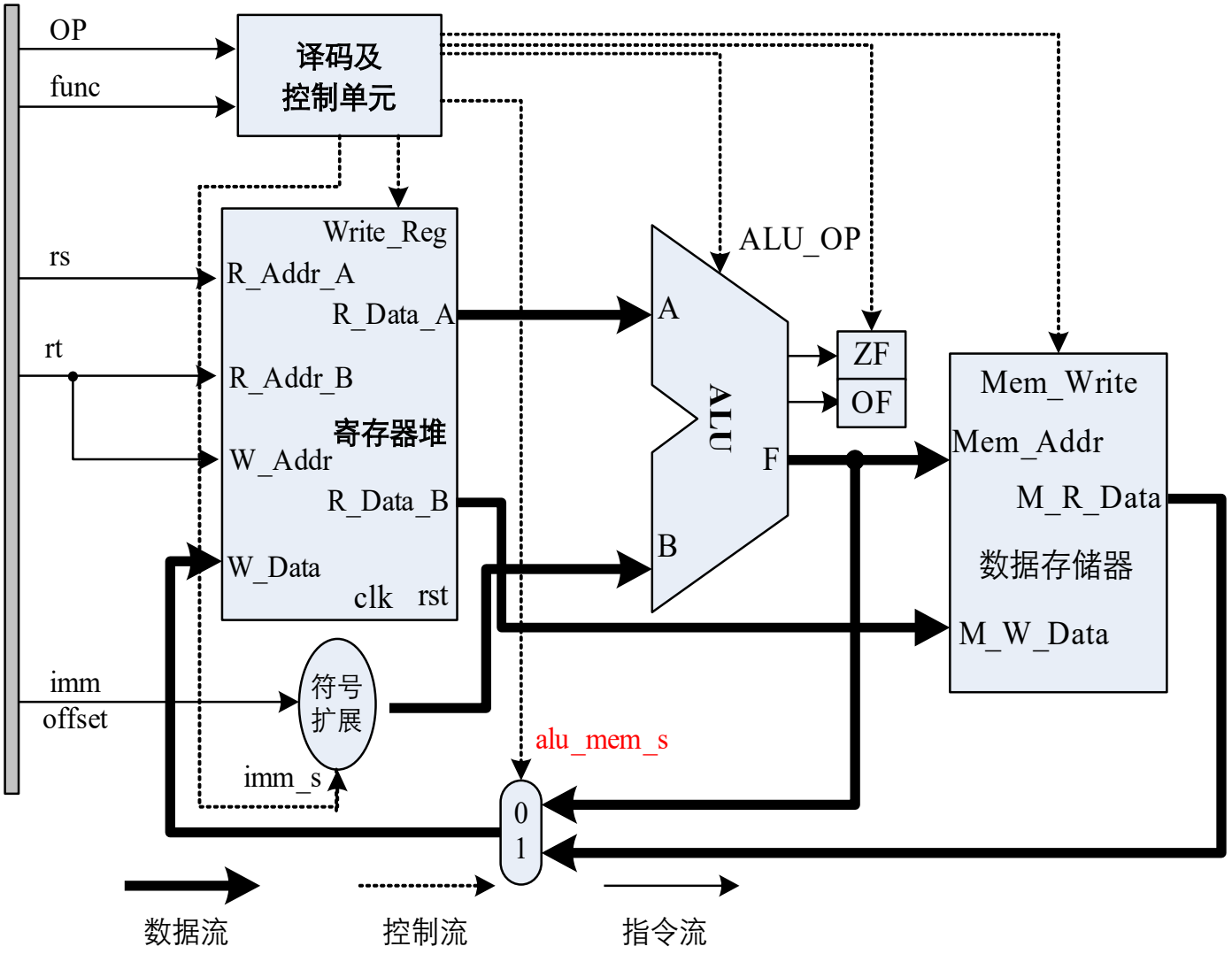
6.1 MIPS I 型存储器访问指令格式及编码

字段	OP	rs	rt	offset	功能描述
位数	6	5	5	16	
汇编助记符	编码				
lw rt, offset(rs)	100011	rs	rt	offset	取数： (rt+offset) →rt
sw rt, offset(rs)	101011	rs	rt	offset	存数： rt→(rs+offset)

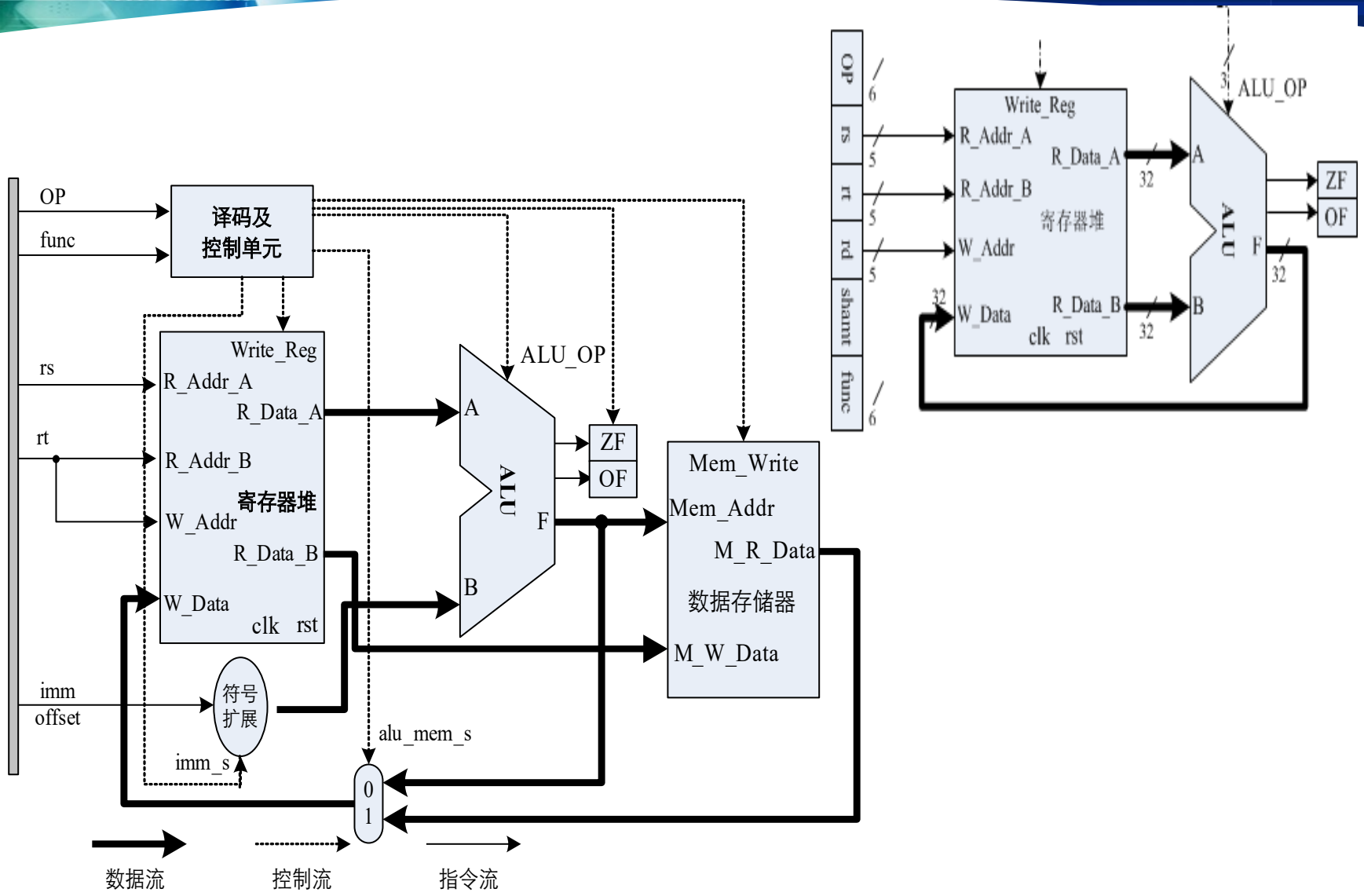
执行 I 型访存指令的功能部件和数据通路



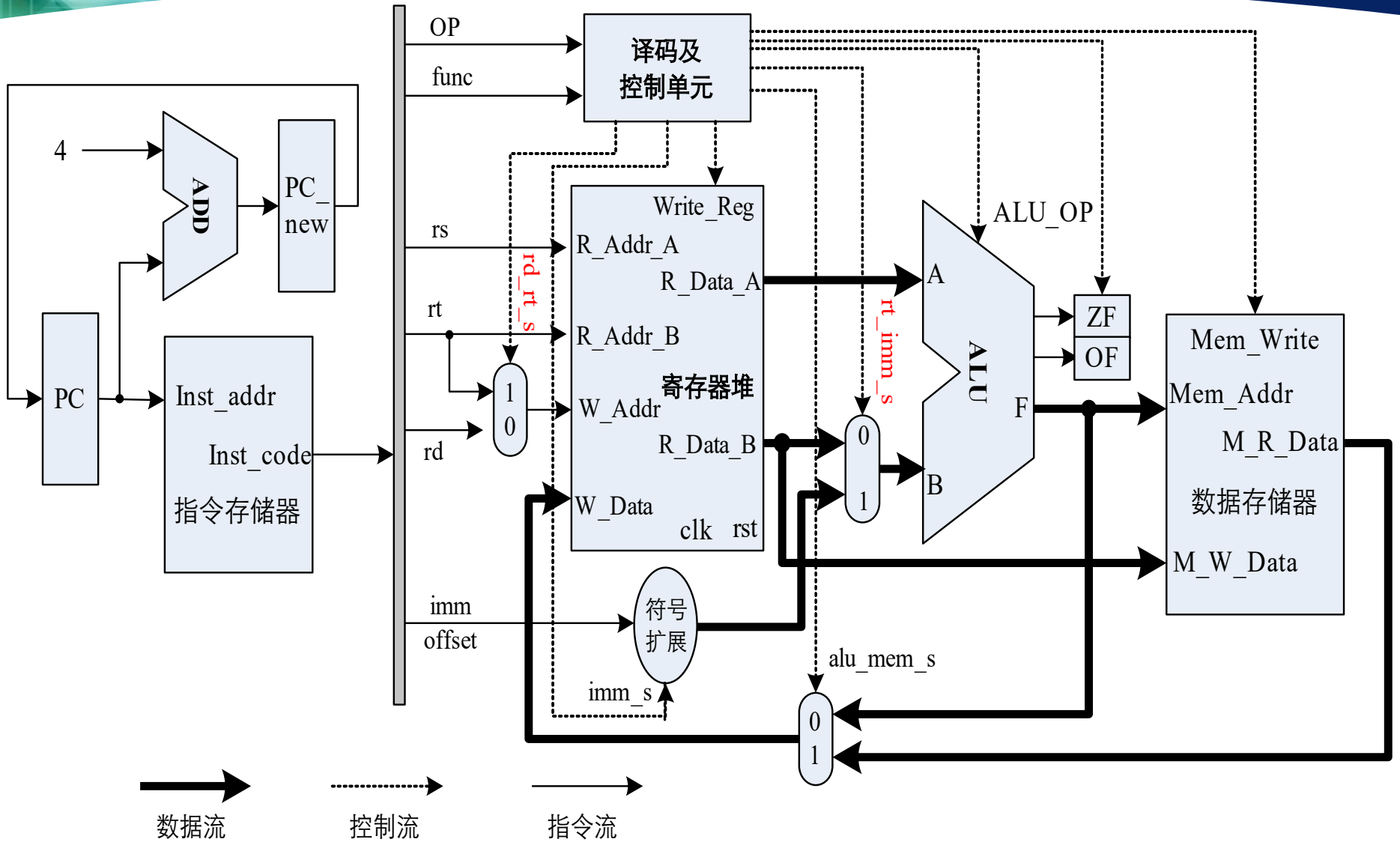
执行 I 型指令的功能部件和数据通路



执行 R-I 型指令的数据通路



执行 R-I 型指令的数据通路



R-I 系统结构下，R 指令执行过程分析

- add $t1,$t2,$t3 （单周期执行过程）
  - 取指令：根据 PC 值，在 clk 上跳沿，从指令存储器中取地址为 PC 的指令 Inst_Code,PC_NEW=PC+4
  - 取操作数：从 MIPS_REGS 中取出 $t2 和 $t3 的值到 ALU 的 A 和 B （rt_imm_s=0）
  - 执行指令：完成 F=A+B，输出 OF 和 ZF (ALU_OP)
  - 送结果：在 clk 下跳沿，将结果 F 送到从 MIPS_REGS 的 $t1(Write_Reg=1， alu_mem_s=0， rd_rt_s=0)，更新标志位 ZF(Set_ZF=1)，更新标志位 OF(Set_OF=1)，更新 PC<=PC_NEW
- 如果 rst=1，复位信号有效，则 PC 清 0，所有寄存器清 0，ZF 和 OF 清 0

R-I 系统结构下，I 型立即数指令执行过程

- addi $t1,$t2,100 （单周期执行过程）
  - 取指令：根据 PC 值，在 clk 上跳沿，从指令存储器中取地址为 PC 的指令 Inst_Code,PC_NEW=PC+4
  - 取操作数：从 MIPS_REGS 中取出 $t2 的值到 ALU 的 A，100 符号扩展（imm_s=1）到 ALU 的 B （rt_imm_s=1）
  - 执行指令：完成 F=A+B，输出 OF 和 ZF (ALU_OP)
  - 送结果：在 clk 下跳沿，将结果 F 送到从 MIPS_REGS 的 $t1(Write_Reg=1， alu_mem_s=0， rd_rt_s=1), 更新标志位 ZF(Set_ZF=1)，更新标志位 OF(Set_OF=1)，更新 PC<=PC_NEW
- 如果 rst=1，复位信号有效，则 PC 清 0，所有寄存器清 0， ZF 和 OF 清 0

R-I 系统结构下，lw 指令执行过程

- lw $t1,100 （ $t2 ） （单周期执行过程）
  - 取指令：根据 PC 值，在 clk 上跳沿，从指令存储器中取地址为 PC 的指令 Inst_Code,PC_NEW=PC+4
  - 计算 EA: 从 MIPS_REGS 中取出 $t2 的值到 ALU 的 A， 100 符号扩展（ imm_s=1 ）到 ALU 的 B （ rt_imm_s=1 ），完成 F=A+B (ALU_OP)， EA
  - 执行指令：在 clk 下跳沿，到 RAM 中取数据 memory[EA] 送到 MIPS_REGS 的 $t1(Write_Reg=1， alu_mem_s=1， rd_rt_s=1), 不更新标志位 (Set_ZF=0， Set_OF=0)，更新 PC<=PC_NEW
- 如果 rst=1，复位信号有效，则 PC 清 0，所有寄存器清 0， ZF 和 OF 清 0

R-I 系统结构下， sw 指令执行过程

- sw $t1,100 （ $t2 ） （单周期执行过程）
  - 取指令：根据 PC 值，在 clk 上跳沿，从指令存储器中取地址为 PC 的指令 Inst_Code,PC_NEW=PC+4
  - 计算 EA: 从 MIPS_REGS 中取出 $t2 的值到 ALU 的 A， 100 符号扩展（ imm_s=1 ）到 ALU 的 B （ rt_imm_s=1 ），完成 F=A+B (ALU_OP)， EA
  - 执行指令：在 clk 下跳沿，MIPS_REGS 的 $t1 的数据（ R_Data_B) 送到 RAM 的 memory[EA] (Mem_Wirte=1), 不更新标志位 (Set_ZF=0， Set_OF=0)，更新 PC<=PC_NEW
- 如果 rst=1，复位信号有效，则 PC 清 0，所有寄存器清 0， ZF 和 OF 清 0

第 4 章 实验系统

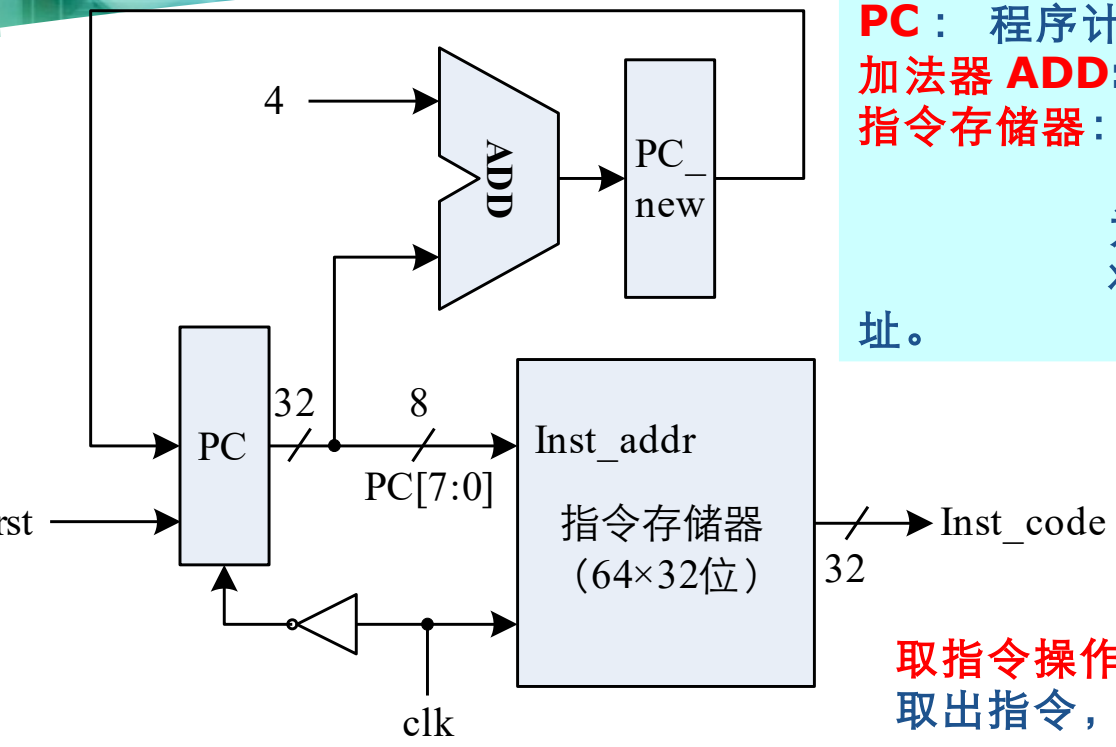
实验七 取指令实验

实验八　实现R型指令的CPU设计

实验九　实现R-I型指令的CPU设计

实验十　实现R-I-J型指令的CPU设计

实验七 取指令与指令译码实验



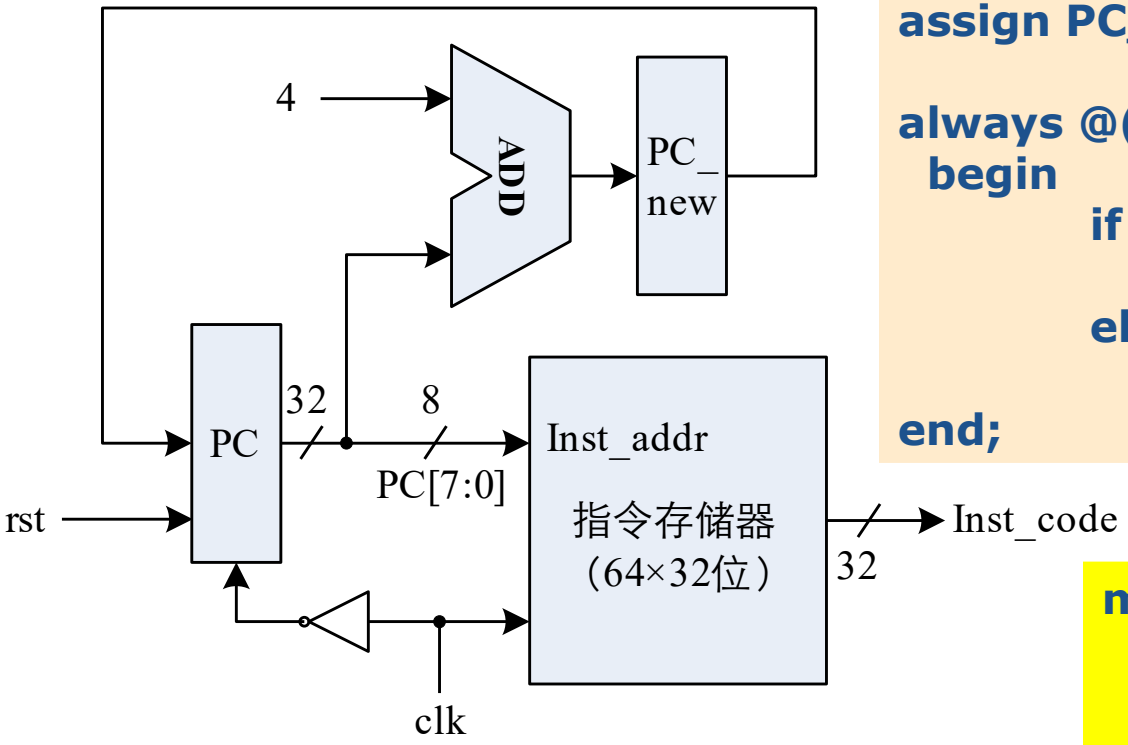
PC： 程序计数器，当前指令的地址
加法器 ADD: 完成 **PC+4,** 下一条指令的地址
指令存储器： **ROM_B,** 采用 **Memory IP** 核实现

为简单起见，指令存储器 **:64*32** 位，将 **PC** 的低 **8** 位作为指令存储器的地址。

取指令操作：根据 **PC** 内容到指令存储器中取出指令， 然后 **PC+4→PC** 。

```
ROM_B Inst_ROM (                          // 采用 Memory IP 核实现
          .clka(clk), // input clka
          .addra(PC[7:2]), // input [5 : 0] addra
          .douta(Inst_code) // output [31 : 0] douta
        );
```

实验七 取指令与指令译码实验




```
assign PC_new = PC + 4;

always @(negedge clk or posedge rst)
  begin
          if (rst)
                  PC <= 32'h0000_0000;
          else
                  PC <= PC_new;
end;
```

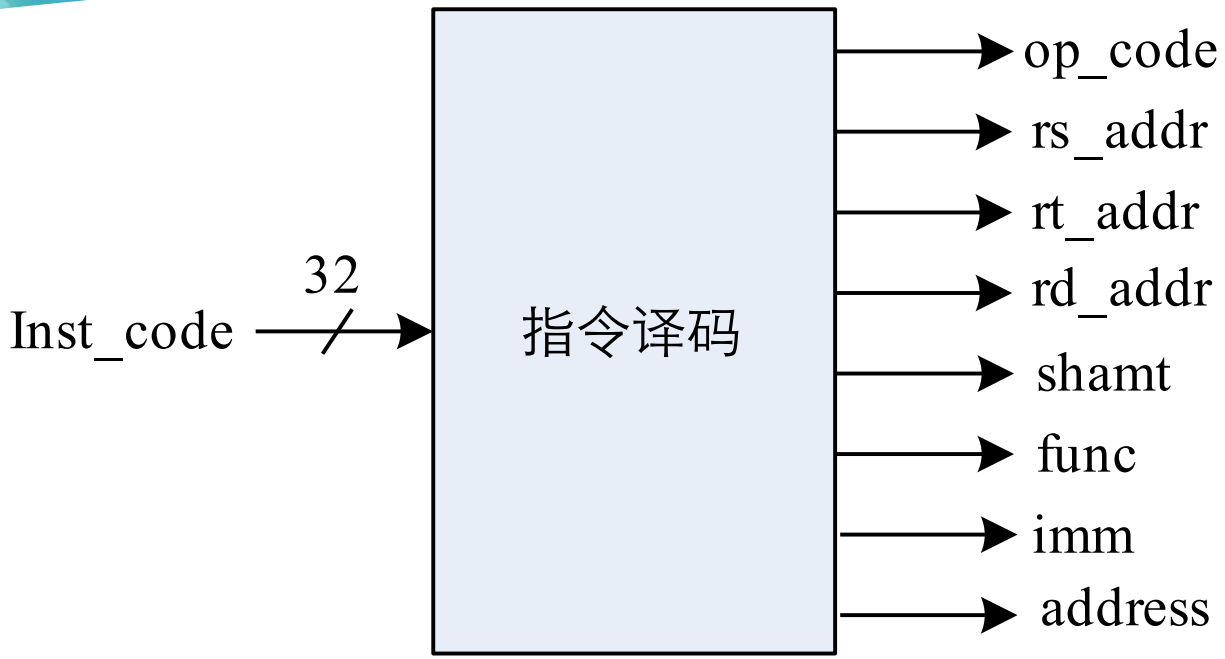
```
module IF_M(
    input clk,
    input rst,
    output [31:0] Inst_code
)
```

单周期 MIPS CPU （即每个周期执行一条指令）

在指令周期（即时钟周期） clk 上跳沿，执行取指令操作，在 clk 下跳沿更新 PC 值

添加了复位信号 rst，当 rst=1 时， PC 清零，即指定 MIPS CPU 从 0 号主存开始执行程序。

实验七 取指令与指令译码实验




```
assign op_code =  Inst_code[31:26];
assign rs_addr =  Inst_code[25:21];
assign rt_addr =  Inst_code[20:16];
assign rd_addr=  Inst_code[15:11];
assign shamt = Inst_code[10:6];
assign func =  Inst_code[5:0];
assign imm= Inst_code[15:0];
assign address = Inst_code[15:0];
```

其实不是真的译码功能

实验要求

（1） 在 ISE 中使用 Memory IP 核生成一个 Inst_ROM，当做指令
存储器，并关联一个实验六所生成的 *.coe 文件。
（2） 编程实现取指令模块，调用 Inst_ROM 指令存储器模块。
（3） 编写一个实验验证的顶层模块，可按照以下方法设计实验，也可
以自行设计验证实验。
a) 一个按钮提供 rst； 1 个按钮提供 clk；
b) 2 位开关作为指令码显示的选择控制：用于选择读出的 32 位 指令码 的某个字节到 8 位 LED 灯显示；

在板卡上先按动 rst 按钮使 PC 清零，按动 clk 按钮则读取指令，拨动 2 位逻辑开关观察并记录读出的 32 位指令代码；再次按动 clk 按钮，则读取第二条指令，观察并记录

思考与探索

顶层模块参考

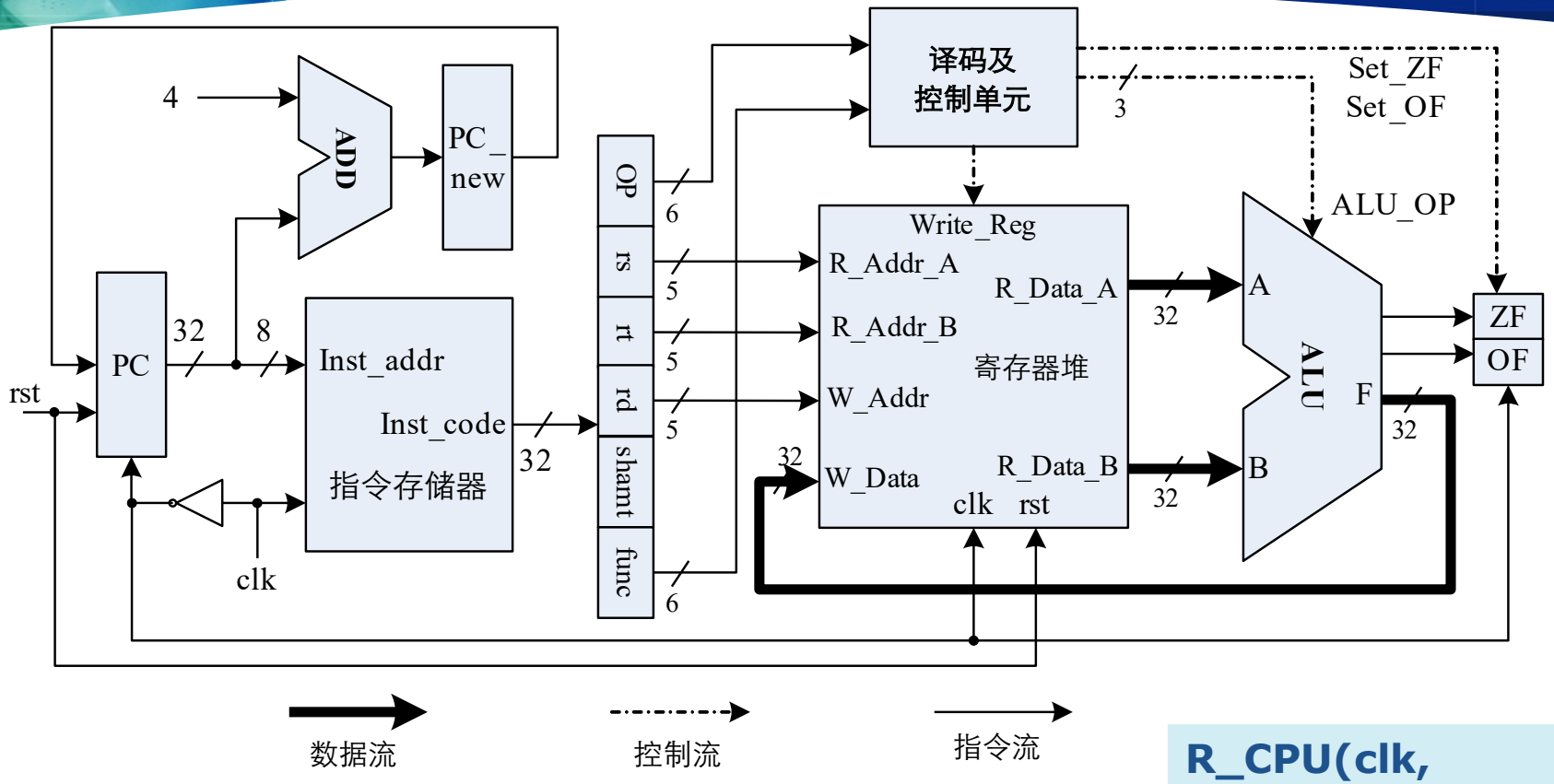
```
module IF_Exp(
        input clk,
        input rst,
        input [1:0] SW,
        output reg [7:0] LED
    );

        wire [31:0] Inst_code;

        IF_M    My_CPU_IF(clk,rst,Inst_code);
        LED_DISPLAY disp_IF(SW, Inst_code,LED);

endmodule
```

根据两位开关 SW 选择 Inst_code 的字节送 LED 输出

实验八 实现 R 型指令的 CPU 设计实验

字段	OP	rs	rt	rd	shamt	func	功能描述
位数	6	5	5	5	5	6	
汇编助记符	编码						
add rd,rs,rt	000000	rs	rt	rd	00000	100000	算术加：
sub rd,rs,rt	000000	rs	rt	rd	00000	100010	算术减：
and rd,rs,rt	000000	rs	rt	rd	00000	100100	逻辑与：
or rd,rs,rt	000000	rs	rt	rd	00000	100101	逻辑或：
xor rd,rs,rt	000000	rs	rt	rd	00000	100110	逻辑异或： ⊕ rt
nor rd,rs,rt	000000	rs	rt	rd	00000	100111	逻辑或非：
sltu rd,rs,rt	000000	rs	rt	rd	00000	101011	无符号数小于则置位： if (rs < rt) rd=1 else rd=0

实验八 实现 R 型指令的 CPU 设计实验



R_CPU(clk,
rst,
FR_ZF,
FR_OF,
ALU_F)

取指令模块 IF_M (调用 ROM_B 模块）
运算器模块 ALUREG(调用 ALU 模块和 MIPS_REG 模块）
译码及控制单元模块
（产生控制信号： ALU_OP, Set_ZF,Set_OF, Write_Reg)
标志位赋值（根据控制信号 Set_ZF,Set_OP ，在 clk 下降沿赋值）

时序问题

- 如何在一个时钟脉冲周期内按序完成取指令、分析指令和执行指令?
- 分析哪些部件需要时钟脉冲同步，哪些部件不需要时钟脉冲。
  - 指令存储器的读操作、 PC 值的更新、寄存器的写操作及标志寄存器的更新需要 clk
  - 寄存器的读操作、 ALU 的运算操作等皆不需要 clk ，可视为组合逻辑电路。
- 设计时序:
  - 在 clk 的上升沿，启动指令存储器依据 PC 读出指令;
  - 在 clk 高电平持续期间，可以完成 PC 值的自增、指令译码、寄存器读操作，随后完成 ALU 运算;
  - 在 clk 的下降沿则完成目的寄存器的写入、 PC 值的更新和标志寄存器的更新。
- 因此，将 clk 反相后作为寄存器堆、 PC 和标志寄存器的打入脉冲

标志位寄存器赋值（CLK 下降沿）

- ZF 和 OF 为 ALU 输出
- FR_ZF 和 FR_OF 为标志寄存器的值

```
always @(posedge rst or negedge clk)
begin
   if (rst)
         … //FR_ZF， FR_OF 复位；
   else
     begin
         if (Set_ZF) FR_ZF <= ZF;
         if (Set_OF) FR_OF <= OF;
     end
end
```

指令对标志寄存器的影响一般遵循以下规律：

（1）传送类指令和跳转类指令不影响标志位；

（2）有符号算术运算类指令（包括 slt 和算术移位指令）影响 ZF 和 OF；

（3）无符号算术运算类指令和逻辑运算类指令影响 ZF，不影响 OF；

（4）条件转移类指令一般会使用标志位 ZF。

根据规则，在译码和控制模块设置变量表示是否在 CLK 的下降沿更新标志寄存器，即 Set_ZF 和 Set_OF

译码和控制

func 字段与 ALU_OP 的编码对照

指令	func	ALU_OP	操作
add rd,rs,rt	100000	100	逻辑与
sub rd,rs,rt	100010	101	逻辑或
and rd,rs,rt	100100	000	逻辑异或
or rd,rs,rt	100101	001	逻辑或非
xor rd,rs,rt	100110	010	算术加
nor rd,rs,rt	100111	011	算术减
sltu rd,rs,rt	101011	110	小于置位
sllv rd,rt,rs	000100	111	逻辑左移

译码和控制

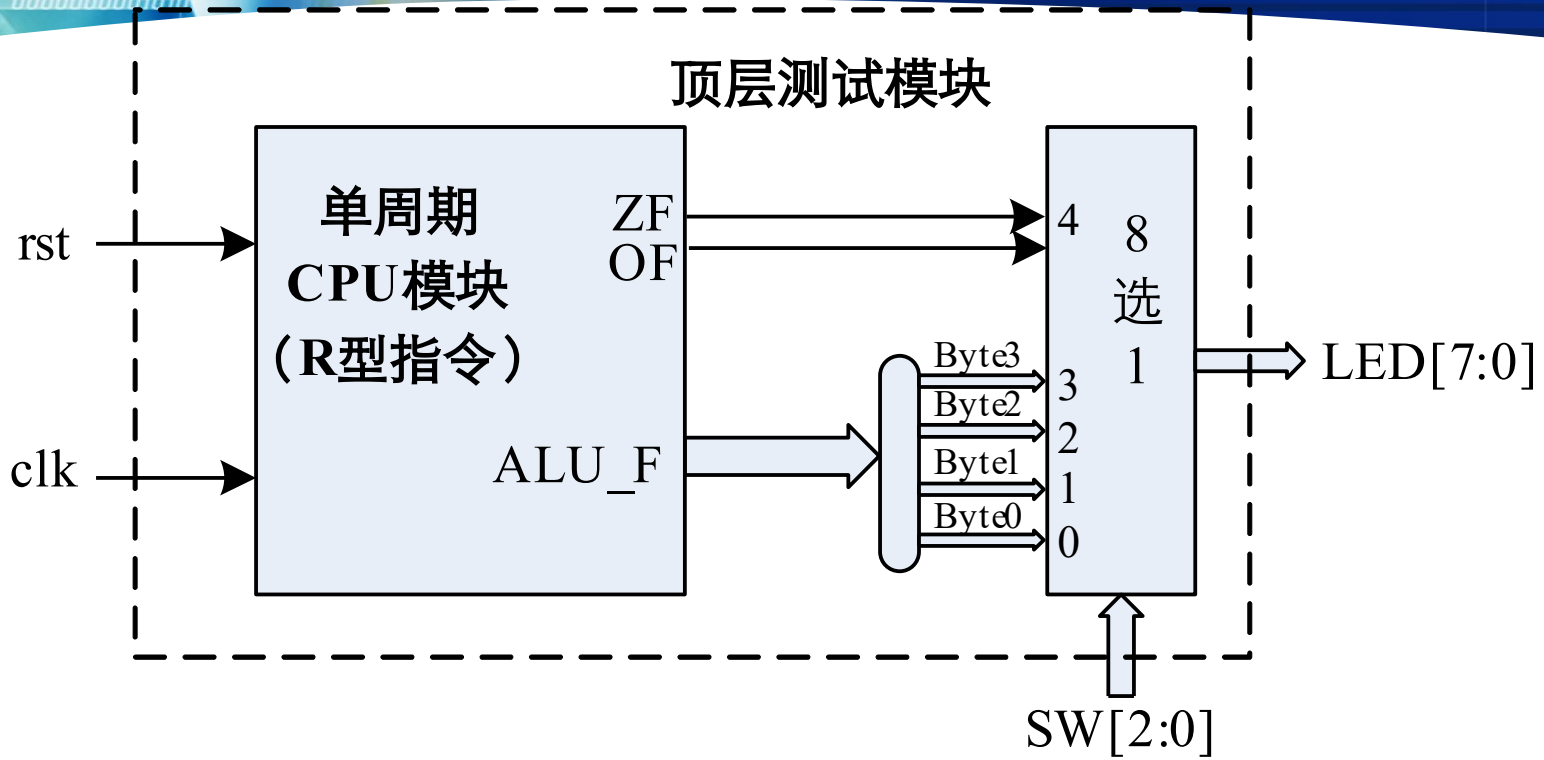
```
assign OP =  Inst_code[31:26];
assign func =  Inst_code[5:0];
。。。。。
```

```
always @(*)
begin
   ALU_OP = 3'b000;
   Write_Reg = 1'b0;
   Set_ZF = 1'b0;
   Set_OF = 1'b0;
   if (OP==6'b000000)    //R 指令
    begin
          Write_Reg = 1'b1;   // 结果送寄存器
          Set_ZF = 1'b1;      // 所有 R 指令影响 ZF
          case (func)
                  。。。。 //ALU_OP 赋值，根据表 6.17
                       // 加法和减法还需 Set_OF 置 1
          endcase
   end
end;
```

实验要求

- （1） 在实验三、实验四、实验五和实验六的基础上，编写一个 CPU 模块，能够实现 8 条指定的 R 型指令。
  - a) 新建一个工程，将实验三实现的 ALU 模块、实验四实现的寄存器模块、实验七实现的指令存储器和取指令模块的 *.v 文件拷贝至工程目录下，并添加到工程中。
  - b) 修改寄存器堆模块，以使 r0 内容恒置全零，且只读。
  - c) 拷贝实验七的指令存储器模块的 ipcore_dir 目录至新工程，并添加 ROM_B.xco 文件；再修改 ROM_B 的初始化关联文件为新工程下的 *.coe 文件 P183
  - d) 新建 CPU 模块，引用 ALU 模块、寄存器堆模块、取指模块实例，并定义一组信号将各模块有序连接。
  - e) 编写 CPU 模块中指令译码、指令执行控制部分的代码，完善 CPU 模块。
- （2） 编写一段测试 8 条指令的汇编程序，使用实验六的汇编器，将其翻译成二进制机器码，并通过关联文件初始化指令存储器。 p185
- （3） 编写一个实验验证的顶层模块

顶层测试模块



rst 可以在约束文件中从一个按钮接入，**clk** 如果接至 **N3** 实验板的 **100MHz**（**V10**）的时钟源，则 **32** 条指令瞬间执行完（每条指令 **10ns**），无法观察运算结果。为此，可以将 **clk** 也使用一个按钮来单步控制，按动一下，执行一条指令，拨动 **3** 位开关，观察结果；再按动一下 **clk** 按钮，则执行下一条……。

注意，clk 要接 FPGA 时钟引脚的 BTND（C9）和 BTNR（D9）

实验九 实现 R-I 型指令的 CPU 设计实验

MIPS I 型立即数寻址指令格式及编码

字段	OP	rs	rt	imm	功能描述
位数	6	5	5	16	
汇编助记符	编码				
addi rt, rs, imm	001000	rs	rt	imm	算术加：rs+imm→rt
andi rt, rs, imm	001100	rs	rt	imm	逻辑与：rs&imm→rt
xori rt, rs, imm	001110	rs	rt	imm	逻辑异或：rs ⊕ imm→rt
sltiu rt, rs, imm	001011	rs	rt	imm	无符号数小于则置位：if (rs < imm) rt=1 else rt=0

imm 符号扩展

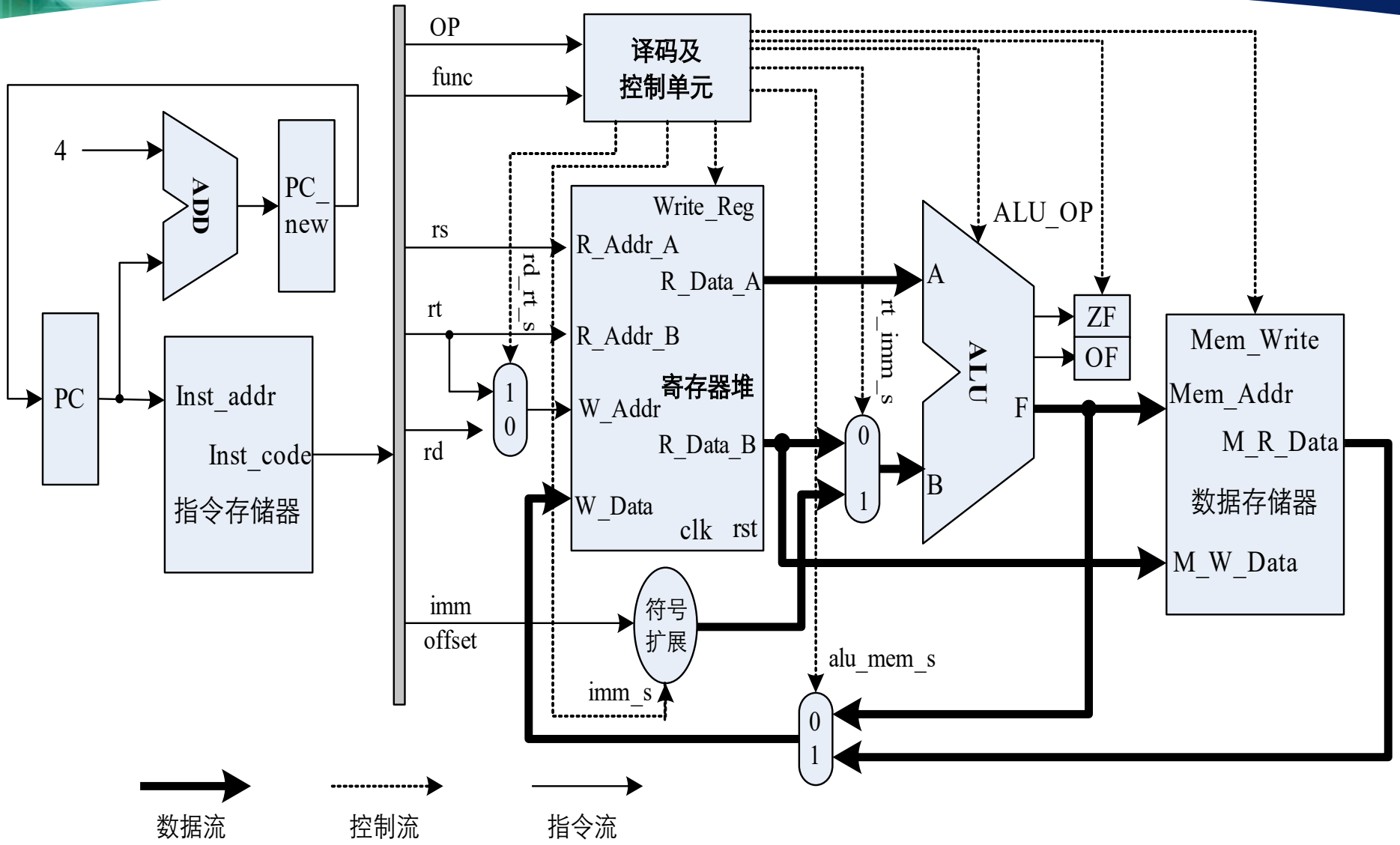
imm 0 扩展

实验九 实现 R-I 型指令的 CPU 设计实验

6.1 MIPS I 型存储器访问指令格式及编码

字段	OP	rs	rt	offset	功能描述
位数	6	5	5	16	
汇编助记符	编码				
lw rt, offset(rs)	100011	rs	rt	offset	取数： (rt+offset) →rt
sw rt, offset(rs)	101011	rs	rt	offset	存数： rt→(rs+offset)

实验九 实现 R-I 型指令的 CPU 设计实验



操作数符号扩展和操作数选择控制

addi rt, rs, imm ； #rs+imm→rt I 指令

add rd,rs,rt; #rs+rt→rd R 指令

- （1） imm 符号扩展和 0 扩展控制 -- imm_s

imm =Instr_Code[15:0], 需扩展为 32 位，有符号数的运算，存 / 取数据的地址偏移量，均需符号扩展；无符号数的运算指令，逻辑运算类指令，则 0 扩展

I 指令，置 imm_s=1 ，则符号扩展， imm_s=0, 则零扩展

R 指令， imm_s 无关

```
assign imm_data = (imm_s)? {16{imm(15)},imm} :
{16{1’b0},imm};
```

- （2） ALU 的操作数（ALU_B）选择控制 -- rt_imm_s

I 指令，置 rt_imm_s=1 ， B 输入扩展后的 32 位 imm ，

R 指令， rt_imm_s =0 ， B 输入 rt 的内容 R_Data_B

```
assign ALU_B = (rt_imm_s)? imm_data : R_Data_B;
```

目的操作数选择控制

addi rt, rs, imm ; #rs+imm→rt I 指令

add rd,rs,rt; #rs+rt→rd R 指令

- （3）目的操作数地址（REG 的 W_Addr）选择控制 -- rd_rt_s

I 指令，置 rd_rt_s=1，选 rt 为目的操作数地址 W_Addr，

R 指令，rd_rt_s=0，选 rd 为目的操作数地址 W_Addr

```
assign W_Addr = (rd_rt_s)? rt:
                                rd;
```

- （4）结果选择控制 （REG 的 W_Data ） - alu_amm_s

I 指令， R 指令，都置 alu_amm_s=0，选 ALU_F 为结果，送 W_Data

```
assign W_Data = alu_mem_s ? M_R_Data :
ALU_F;
```

R-I 指令控制信号

R 指令控制信号

- imm_s 无关
- rt_imm_s =0， B 输入 rt 的内容 R_Data_B
- rd_rt_s=0，选 rd 为目的操作数地址 W_Addr
- 置 alu_amm_s=0，选 ALU_F 为结果
- ALU_OP
- Set_ZF,Set_OF

I 型立即数指令控制信号

- imm_s=1，则符号扩展，imm_s=0,则零扩展
- 置 rt_imm_s=1， B 输入扩展后的 32 位 imm
- 置 rd_rt_s=1，选 rt 为目的操作数地址 W_Addr
- 置 alu_amm_s=0，选 ALU_F 为结果
- ALU_OP
- Set_ZF,Set_OF

lw 和 sw 指令控制信号及输入变量赋值

- lw rt,offset(rs); # (rt+offset) →rt
- sw rt, offset(rs) ; # rt→(rs+offset)
- 公共操作 (取指令，计算 EA)

  ALU_OP=000, ALU 做加法运算

  EA 计算，置 imm_s=1 （符号扩展）

  rt_imm_s=1 （扩展后的 offset 为 ALU 的 B ）

  ALU 运算结果为 RAM 地址， assign Mem_Addr = ALU_F[7:2];
- lw

  置 alu_mem_s =1，存储器数据 M_R_Data 送寄存器 (W_Data)

  R 指令，复位 alu_mem_s ， ALU_F 送寄存器 (W_Data)

  assign W_Data = alu_mem_s ? M_R_Data : ALU_F;

  置 rd_rt_s=1, 选 rt 为目的操作数地址，

  置 Write_Reg=1,W_Data 写入 $rt

lw 和 sw 指令控制信号

lw

- ALU_OP=000
- Set_ZF=0,Set_OF=0
- imm_s=1 （符号扩展）
- rt_imm_s=1 （扩展后的 offset 为 B ）
- alu_mem_s =1，存储器数据 M_R_Data 送寄存器 (W_Data)
- rd_rt_s=1, 选 rt 为目的操作数地址
- Write_Reg=1,W_Data 写入 $rt

sw

- ALU_OP=000
- Set_ZF=0,Set_OF=0
- imm_s=1 （符号扩展）
- rt_imm_s=1 （扩展后的 offset 为 B ）
- Mem_Write=1, $rt 数据写入存储器

实验九 实现 R-I 型指令的 CPU 设计实验

6.1 R-I 型指令的控制流

指令	rd_rt_s	imm_s	rt_imm_s	alu_mem_s	ALU_OP	WriteReg	Mem_Write
add rd,rs,rt	0	——	0	0	100	1	0
sub rd,rs,rt	0	——	0	0	101	1	0
and rd,rs,rt	0	——	0	0	000	1	0
or rd,rs,rt	0	——	0	0	001	1	0
xor rd,rs,rt	0	——	0	0	010	1	0
nor rd,rs,rt	0	——	0	0	011	1	0
sltu rd,rs,rt	0	——	0	0	110	1	0
sllv rd,rs,rt	0	——	0	0	111	1	0
addi rt,rs,imm	1	1	1	0	100	1	0
andi rt, rs, imm	1	0	1	0	000	1	0
xori rt, rs, imm	1	0	1	0	010	1	0
sltiu rt, rs, imm	1	0	1	0	110	1	0
lw rt, offset(rs)	1	1	1	1	——	1	0
sw rt, offset(rs)	——	1	1	——	——	0	1

译码和控制

```
always @(*)
    begin
        ALU_OP = 3'b100;        // 默认做加法
        Write_Reg = 1'b1;       // 默认写寄存器
        Set_ZF = 1'b1;          // 默认影响结果零标志位
        Set_OF = 1'b0;          // 默认不影响溢出标志位
        rd_rt_s = 1'b0;         // 默认写入 rd 指明的寄存器
        imm_s = 1'b0;           // 默认对立即数 / 偏移量进行 0 扩
展
        rt_imm_s = 1'b0;        // 默认读出 rt 寄存器的数据送
ALU_B
        alu_mem_s = 1'b0;       // 默认 ALU 运算结果写入目的寄
存器    Mem_Write = 1'b0;       // 默认不写存储器
        if (OP==6'b000000)
                // 根据 func 确定 ALU_OP, Set_ZF ,   Set_OF
        else
                // 根据 OP 确定各控制信号
```

赋值语句

- assign Reg_W_Addr = (rd_rt_s) ? Rt : Rd;
-
- assign imm_Data = (imm_s) ? {{16{imm[15]}},imm}:{{16{1'b0}},imm};
- assign ALU_B = (rt_imm_s) ? imm_Data : Reg_Data_B;
- assign Reg_W_Data = (alu_mem_s) ? Mem_R_Data : ALU_F;
- assign Mem_Addr = ALU_F[7:2];
- assign Mem_W_Data = Reg_Data_B;

I 型指令的时序

- 对于立即数寻址的 I 型指令，执行的时序同 R 型指令：
  - 在 clk 的上跳沿，指令存储器执行读操作；
  - 在 clk 正脉冲内，读出的指令经过译码、执行运算；
  - 在 clk 的下跳沿，将运算结果打入目的寄存器 rd 或者 rt。
- 对于取数 / 存数指令，对数据存储器的读和写访问都要与 clk 脉冲同步。如果使用 CPU 的 clk 作为数据存储器的 clk，则在 clk 的正边沿来临时，指令还未读出，存储器的有效地址 EA 还未正确产生，因此此时读写存储器不能得到应有的正确结果。因此，可以考虑使用频率更高的时钟作为数据存储器的 clk，频率至少是 CPU 频率的 2 倍以上。
- 时钟设计?

测试程序

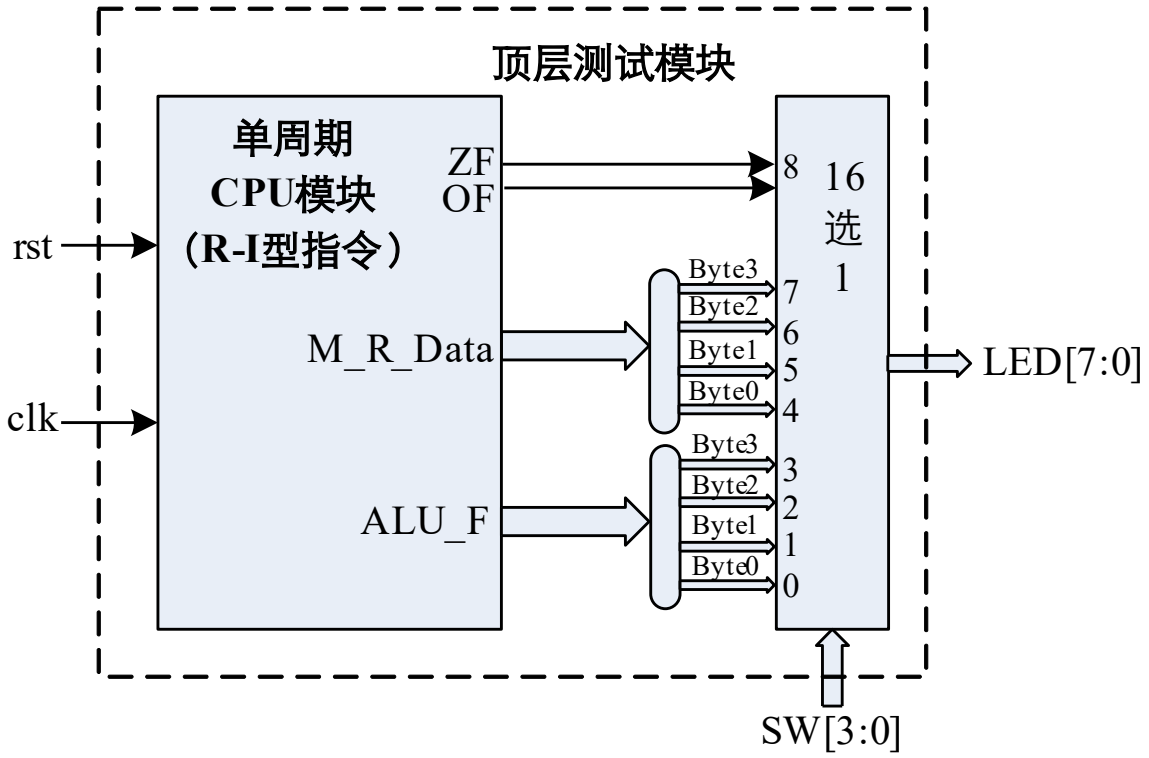
- P193-P194
- 也可以自行设计

实验要求

- （1）在实验八的基础上，编写一个 CPU 模块，除了能够实现实验八的 8 条 R 型指令外，还要求能够实现新的 6 条 I 型指令，见实验内容。
  - a) 将实验八的工程拷贝至新目录下，成为一个新工程；修改 ROM_B 的初始化关联文件为新工程下的 *.coe 文件。
  - b) 在新工程中，使用存储器 IP 核创建一个新的 RAM 模块 RAM_B，指定一个新的 *.coe 文件作为它的初始化关联文件。
  - c) 在 CPU 模块中，引用 RAM_B 的实例，作为数据存储器。
  - d) 定义一些控制信号和数据信号，添加 3 个二选一通道和一个立即数 / 偏移量的扩展部件，然后重新对各模块进行逻辑连接。
  - e) 修改和扩充 CPU 模块中指令译码、指令执行控制部分的代码，完善 CPU 模块。

顶层测试模块

（2） 编写一个实验验证的顶层模块，如图 6.37，也可以自行设计验证实验。



在板卡上先按动 rst 按钮使 PC 清零、寄存器堆和标志 ZF、OF 清零；然后，每按动一次 clk 按钮，就执行一条指令；拨动 4 位逻辑开关，观察并记录指令执行的结果和标志

注意，clk 要接 FPGA 时钟引脚的 BTND（C9）和 BTNR（D9）

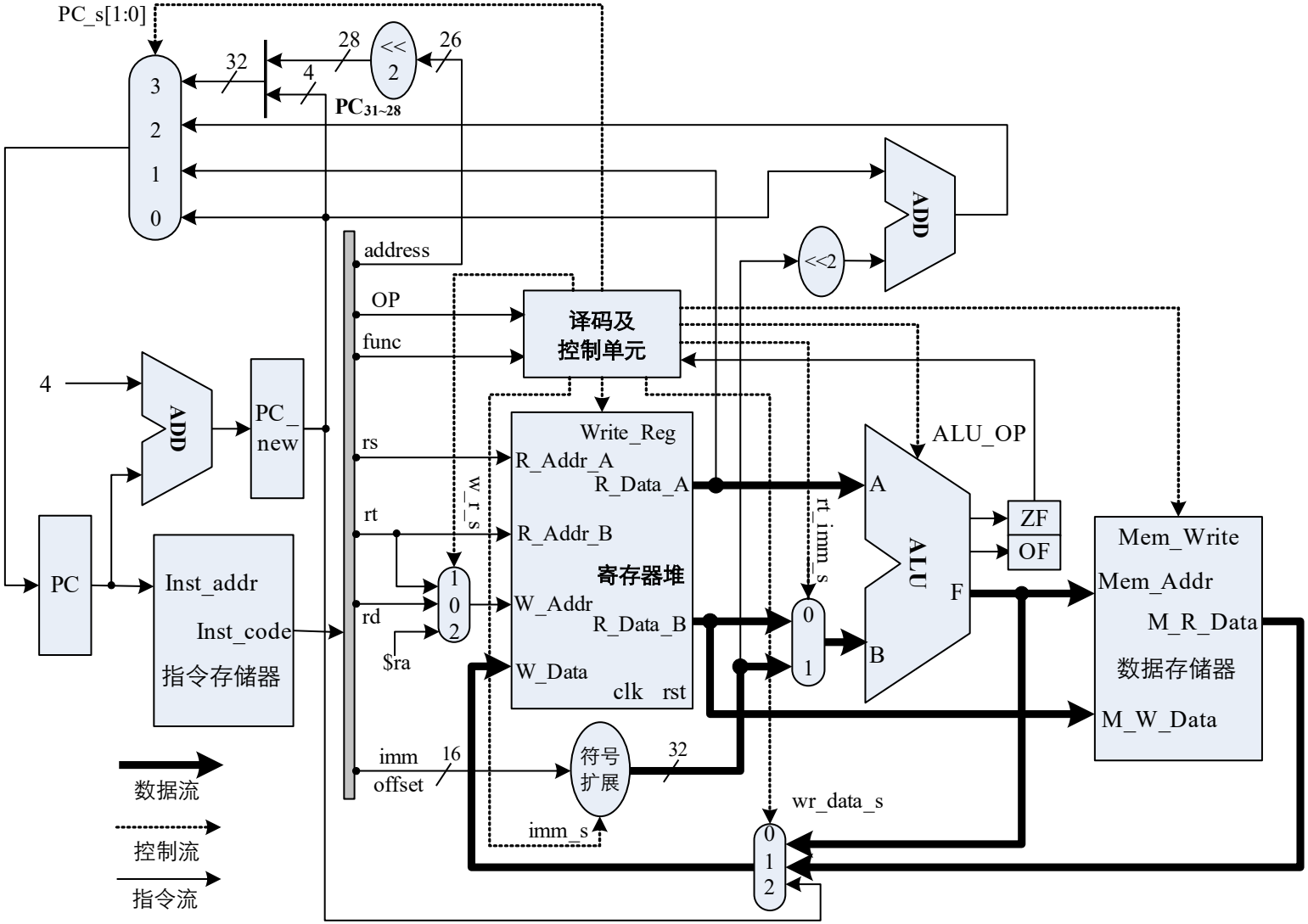
实验要求

- （2） 编写一个实验验证的顶层模块，如图 6.37，也可以自行设计验证实验。
  - a）一个按钮提供 rst； 1 个按钮提供 clk；
  - b) 4 位开关作为当前指令运算结果与存储器读出数据显示的选择控制：
    - 4’b0000~4’b0011 用于选择 ALU 的 32 位运算结果的第 0~3 字节到 8 位 LED 灯显示；
    - 4’b0100~4’b0111 用于选择存储器读出的 32 位数据的第 0~3 字节到 8 位 LED 灯显示；
    - 4’b1000 用于选择 8 位 LED 显示 OF 和 ZF 标志。
  - c) 8 位 LED 灯用于显示当前指令运算结果的字节或标志；

实验十 实现 R-I-J 型指令的 CPU 设计实验

R	字段	OP	rs	rt	rd	sham t	func	功能描述
	位数	6	5	5	5	5	6	
jr rs		000000	rs	00000	00000	00000	001000	无条件跳转： rs
I	字段	OP	rs	rt	offset			功能描述
	位数	6	5	5	16			
beq rs, rt, label		000100	rs	rt	offset			相等转移：
bne rs, rt, label		000101	rs	rt	offset			不相等转移：
J	字段	OP	address					功能描述
	位数	6	26					
j label		000010	address					无条件跳转：
jal label		000011	address					无条件跳转并链接：(PC+4)→$31, {(PC+4)高 4位 ,address,0,0}→PC

实验十 实现 R-I-J 型指令的 CPU 设计实验



指令	w_r_s	imm_s	rt_imm_s	wr_datas	ALU_OP	Write_Rega	Mem_Write	PC_s
add rd,rs,rt	00	——	0	00	100	1	0	00
sub rd,rs,rt	00	——	0	00	101	1	0	00
and rd,rs,rt	00	——	0	00	000	1	0	00
or rd,rs,rt	00	——	0	00	001	1	0	00
xor rd,rs,rt	00	——	0	00	010	1	0	00
nor rd,rs,rt	00	——	0	00	011	1	0	00
sltu rd,rs,rt	00	——	0	00	110	1	0	00
sllv rd,rs,rt	00	——	0	00	111	1	0	00
addi rt,rs,imm	01	1	1	00	100	1	0	00
andi rt, rs, imm	01	0	1	00	000	1	0	00
xori rt, rs, imm	01	0	1	00	010	1	0	00
sltiu rt, rs, imm	01	0	1	00	110	1	0	00
lw rt, offset(rs)	01	1	1	01	——	1	0	00
sw rt, offset(rs)	——	1	1	——	——	0	1	00
jr rs	——	——	——	——	——	0	0	01
beq rs, rt, label	——	——	0	——	101	0	0	00/10
bne rs, rt, label	——	——	0	——	101	0	0	00/10
j label	——	——	——	——	——	0	0	11

The End!