单周期CPU的控制器及顶层设计

1. 控制器的设计

2.1. 关于MIPS指令类型

类型	31_26位	25_21位	20_16位	15_11位	10_6位	5_0位
R	操作码	源操作数1(rs)	源操作数2(rt)	目的操作数(rd)	移位位数	功能码
I(立即数)	操作码	源操作数1(rs)	目的操作数(rt)	立即数	立即数	立即数
J(跳转)	操作码	立即数	立即数	立即数	立即数	立即数

1️⃣对于R型指令

1. 31_26位全0(special类型)，需要再根据5_0位决定指令类型
2. 对于R型指令，10_6位为移位位数，当指令不涉及位移时全部置0

2️⃣对于I型指令：直接由5_0位决定指令类型

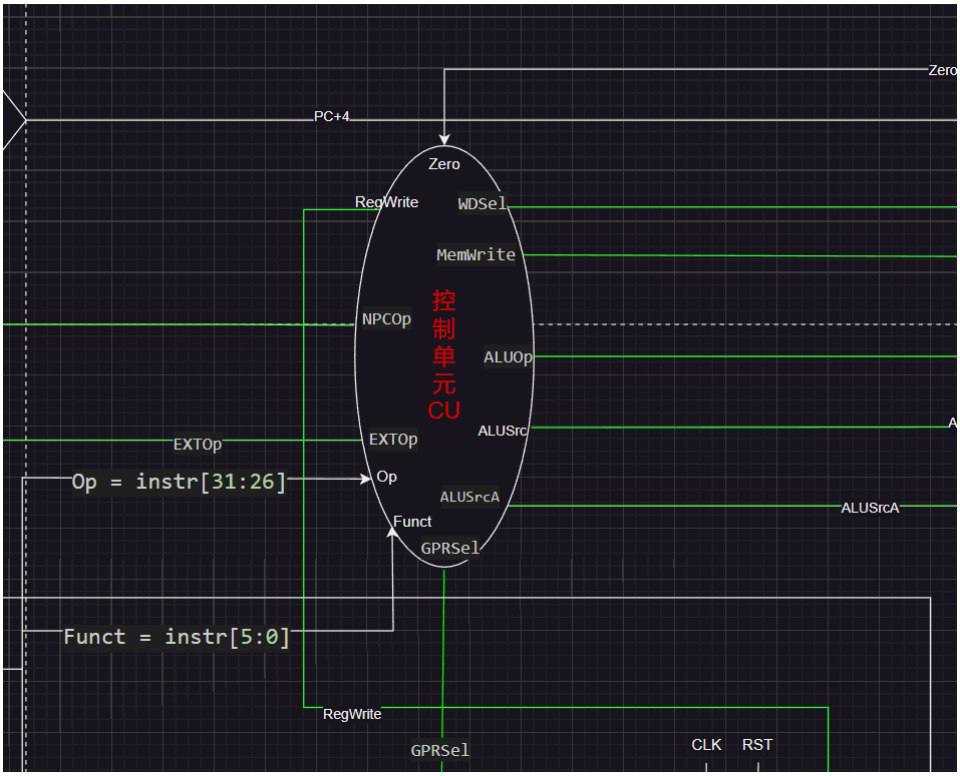
3️⃣对于J型指令：

1. `j`：无条件跳转
2. `jal`：无条件跳转并链接
3. `jr`：寄存器跳转(其实这个指令是R型的)
4. `jalr`：寄存器跳转并链接(其实这个指令是R型的)

2.2. 模块接口：详见CPU的整体数据通路

信号名	方向	描述
Op	input	操作码，指令的31-26位
Funct	input	功能码，指令的5-0位
Zero	input	判断alu运算的结果是否为0，便于 bne beq的操作
RegWrite	output	寄存器写信号
MemWrite	output	存储器写信号
EXTOp	output	立即数扩展信号，决定立即数是零扩展还是符号扩展
ALUOp	output	ALU控制信号，决定了alu采用何种操作
NPCOp	output	PC控制信号，决定了PC如何更新自己
ALUSrc	output	决定传入ALU的B的值，来自立即数还是寄存器组

信号名	方向	描述
ALUSrcA	output	决定传入ALU的A的值，来自寄存器组还是偏移量
GPRSe1	output	决定将运算结果回送哪个寄存器
WDSe1	output	要将什么数据写回寄存器



2.3. 功能1：确定指令类型

根据Op(操作码)和Funct(功能码)

2.3.1. 对于I型指令

只需通过Op，即可判断其类型，例如

```
wire i_addi = ~Op[5]&~Op[4]& Op[3]&~Op[2]&~Op[1]&~Op[0];
//addi指令的操作码是001000，当Op所代表的指令是addi时，就会有i_addi=1
```

以此类推有：

```
wire i_ori  = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]& Op[0]; // ori
wire i_lw   =  Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0]; // lw
wire i_sw   =  Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]& Op[0]; // sw
wire i_beq  = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]&~Op[0]; // beq
wire i_lui  = ~Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0]; // lui
wire i_slti = ~Op[5]&~Op[4]& Op[3]& ~Op[2]& Op[1]& ~Op[0]; // slti
wire i_bne  = ~Op[5]&~Op[4]& ~Op[3]& Op[2]& ~Op[1]& Op[0]; // bne
wire i_andi = ~Op[5]&~Op[4]& Op[3]& Op[2]& ~Op[1]& ~Op[0]; //andi
```

2.3.2. 对于R型指令

其Op全为0，需要通过Funct来判断其类型，例如

```
//rtype用于检测当前执行的指令是否为R型，|Op将Op的所有位或在一起，当指令为R型时才有~|Op=1
wire rtype  = ~|Op;
//确定了指令为R型后，例如add的功能码为100000，当Funct所指为add时，才会有i_add=1
wire i_add = rtype&
Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0];
```

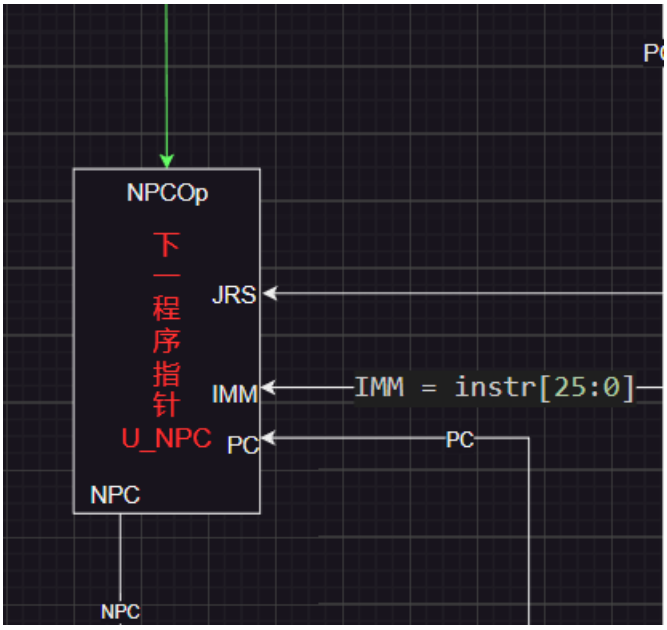
以此类推有：

```
wire i_add  = rtype&
Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; // add
wire i_sub  = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&
Funct[1]&~Funct[0]; // sub
wire i_and  = rtype& Funct[5]&~Funct[4]&~Funct[3]&
Funct[2]&~Funct[1]&~Funct[0]; // and
wire i_or   = rtype& Funct[5]&~Funct[4]&~Funct[3]&
Funct[2]&~Funct[1]& Funct[0]; // or
wire i_slt  = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&
Funct[1]&~Funct[0]; // slt
wire i_sltu = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&
Funct[1]& Funct[0]; // sltu
wire i_addu = rtype&
Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]& Funct[0]; //
addu
wire i_subu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&
Funct[1]& Funct[0]; // subu
wire i_sll  = rtype&
~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; //sll
wire i_nor  = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&
Funct[1]& Funct[0]; //nor
wire i_srl  = rtype&
~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&Funct[1]&~Funct[0]; //srl
wire i_sllv = rtype&
~Funct[5]&~Funct[4]&~Funct[3]&Funct[2]&~Funct[1]&~Funct[0];
 //sllv
wire i_srlv = rtype&
~Funct[5]&~Funct[4]&~Funct[3]&Funct[2]&Funct[1]&~Funct[0];  //srlv
```

2.3.3. 对于跳转指令

```
//j指令的操作码是000010，当Op与000010匹配时，i_j信号被置为1，表示当前指令为j
wire i_j = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]&~Op[0];

//jal指令的操作码是000011，当Op与000011匹配时，i_jal信号被置为1，表示当前指令为jal
wire i_jal = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0];

//jr是一个R型指令，因此首先检查Op是否表示R型指令rtype = ~|Op
//检查Funct字段是否匹配jr指令的Funct码001000，如果匹配，i_jr被置为1
wire i_jr = rtype & ~Funct[5] & ~Funct[4] & Funct[3] & ~Funct[2] & ~Funct[1] & ~Funct[0];

//jalr也是一个R型指令，所以同样先检查Op是否为0
//检查Funct字段是否匹配jalr指令的Funct码001001，如果匹配，i_jalr被置为1
wire i_jalr = rtype & ~Funct[5] & ~Funct[4] & Funct[3] & ~Funct[2] & ~Funct[1] & Funct[0];
```

2.4. 功能2：NPCOp决定PC下一步的行为



1️⃣四路选择机制

条件	对应宏	含义
NPCOp=00	NPC_PLUS4	普通情况下，PC 值加 4，即执行顺序下一条指令
NPCOp=01	NPC_BRANCH	分支指令，PC 值根据分支条件和偏移量计算
NPCOp=10	NPC_JUMP	无条件跳转指令，PC 值直接跳转到指定地址
NPCOp=11	NPC_JUMPR	寄存器跳转指令，PC 值根据寄存器内容跳转

2️⃣有关指令

1. 分支情况

```
//i_beq &  zero表示当执行beq(分支若等于)指令，并且结果为零(相等)时，采用分支逻辑计算PC
//i_beq & ~zero表示当执行bne(分支若不等)指令，并且结果非零(不等)时，采用分支逻辑计算PC
```

2. 寄存器跳转

```
//i_jr 和 i_jalr 是寄存器跳转指令，PC的值将基于寄存器的内容
```

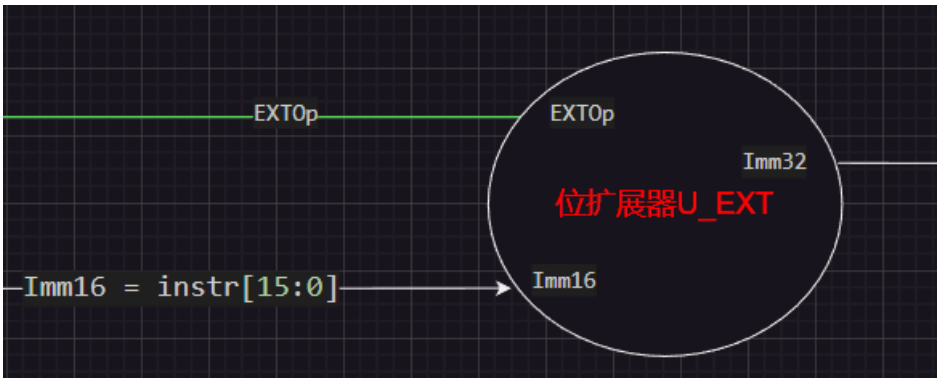
3. 无条件跳转

```
//i_j和i_jal表示无条件跳转/无条件跳转并链接指令，PC直接跳转到指定地址
```

3️⃣代码实现

```
//当为分支跳转or寄存器跳转时，设置NPCOp=x1
assign NPCOp[0] = i_beq & Zero | i_bne & ~Zero | i_jr | i_jalr;

//当为无条件跳转or寄存器跳转时，设置NPCOp=1x
assign NPCOp[1] = i_j | i_jal | i_jr | i_jalr;
```

2.5. 功能3：EXTOp决定立即数的位扩展方式



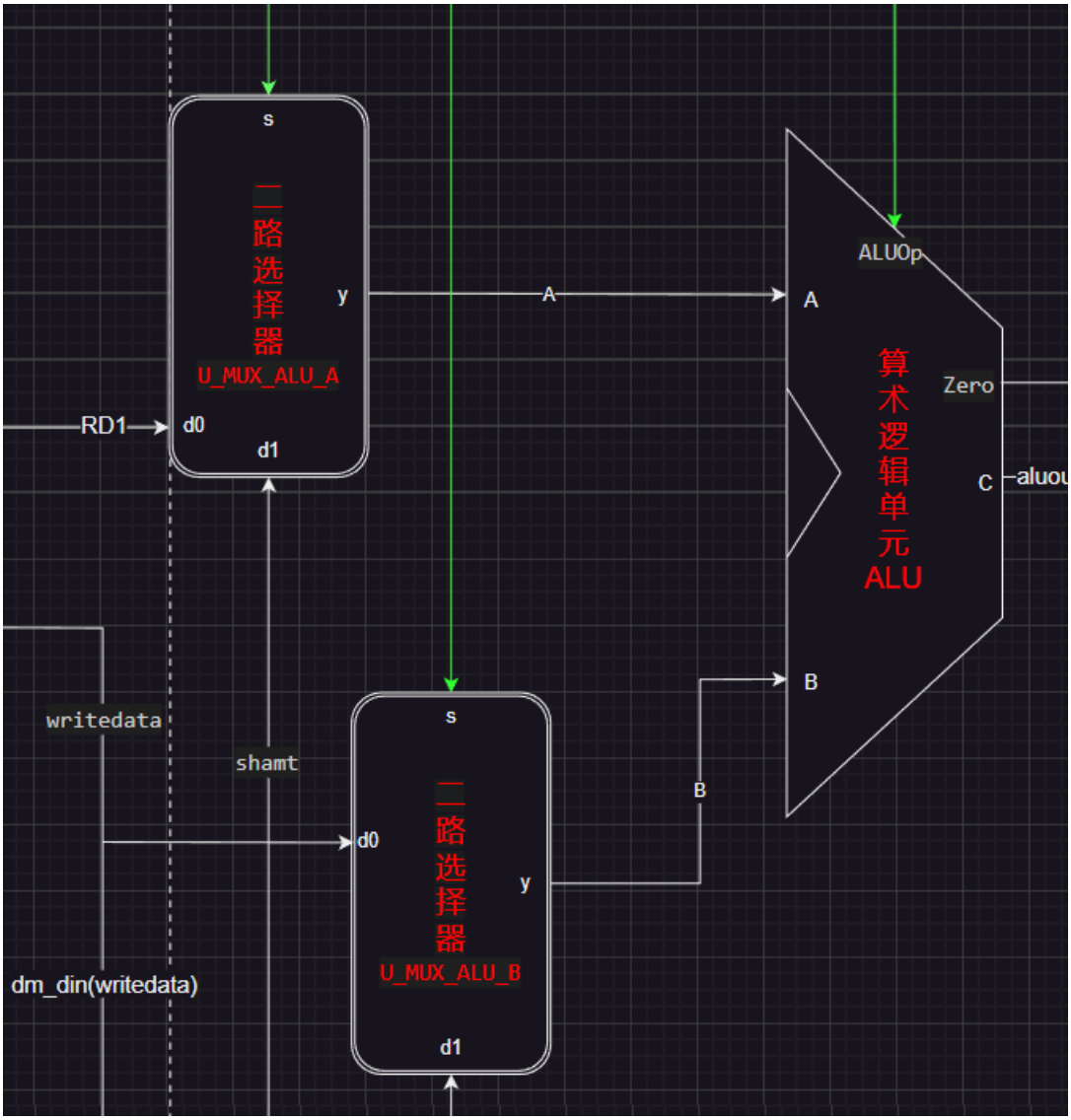
1️⃣二路选择机制

条件	含义
EXTOp=1	进行有符号数的扩展，高位补符号位
EXTOp=0	进行无符号数的扩展，高位补0

2️⃣代码实现：对涉及高位补符号位的指令做如下处理

```
assign EXTOp = i_addi | i_lw | i_sw | i_lui | i_slti; // signed extension
```

2.6. 功能4：决定将什么东西输入ALU



2.6.1. ALUSrc：确定给ALU的B端输入

1️⃣ 二路选择机制

条件	含义
ALUSrc=1	ALU 的 B 操作数从指令的立即数字段获取
ALUSrc=0	ALU 的 B 操作数从寄存器获取

2️⃣ 涉及立即数的一些指令

1. `i_lw`(加载字)/ `i_sw`(存储字)：加载或存储要涉及地址，一般是通过A端输入基地址(来自寄存器)，B端输入立即数来寻址的
2. `i_addi`(立即数加法)/`i_ori`(立即数或运算)/`i_andi`(立即数与运算)：涉及寄存器数和立即数的算术逻辑运算，A端输入寄存器数，B端输入立即数
3. `i_lui`(加载上部立即数)：将一个立即数加载到寄存器的高16位中， B 操作数来自立即数
4. `i_slti`(立即数设立小于)：比较寄存器和一个立即数，寄存器数<立即数则返回1

3️⃣ 代码实现：

```
assign ALUSrc     = i_lw | i_sw | i_addi | i_ori | i_lui | i_slti | i_andi;
```

2.6.2. ALUSrcA：确定给ALU的A端输入

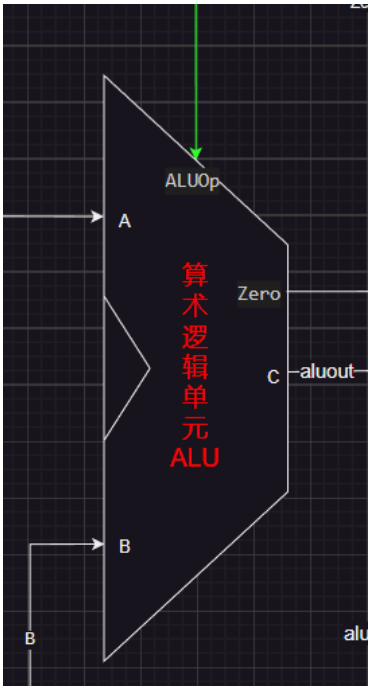
1️⃣二路选择机制

条件	含义
ALUSrcA=1	ALU 的A操作数从指令的偏移量字段
ALUSrcA=0	ALU 的A操作数从寄存器获取

2️⃣代码实现：对设计移位的指令

```
assign ALUSrcA = i_sll | i_srl; // ALU A 来自shamt字段
```

2.7. 功能5：ALUOp决定ALU执行什么操作



1️⃣ALU的控制机制

ALU 操作码	对应宏	含义
4'b0000	ALU_NOP	无操作
4'b0001	ALU_ADD	加法
4'b0010	ALU_SUB	减法
4'b0010	ALU_BNE	分支不等于
4'b0011	ALU_AND	与操作
4'b0011	ALU_ANDI	立即数与操作
4'b0100	ALU_OR	或操作

ALU 操作码	对应宏	含义
4'b0101	ALU_SLT	设立小于
4'b0101	ALU_SLTI	立即数设立小于
4'b0110	ALU_SLTU	无符号设立小于
4'b0111	ALU_SLLV	变量逻辑左移
4'b1000	ALU_SLL	逻辑左移
4'b1001	ALU_NOR	或非操作
4'b1010	ALU_LUI	加载上部立即数
4'b1011	ALU_SRL	逻辑右移
4'b1100	ALU_SRLV	变量逻辑右移

and so on.....

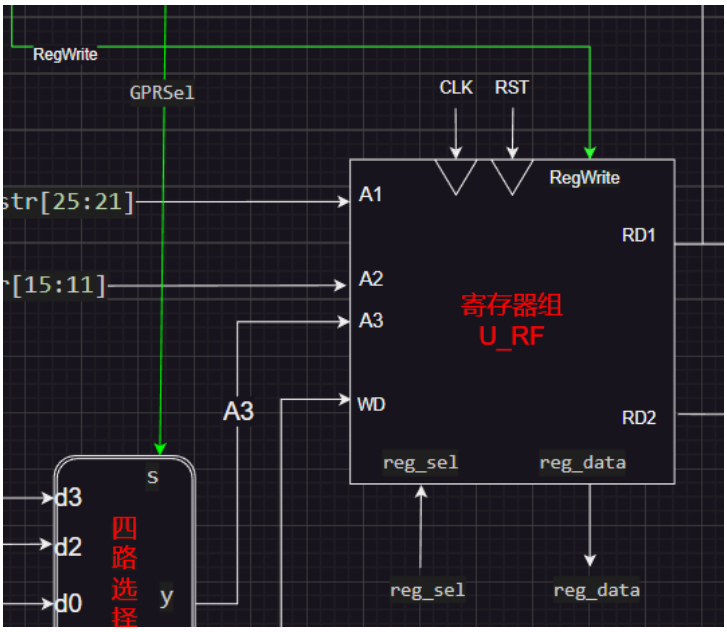
2️⃣代码实现

```
//这些操作对应的ALU操作码都是xxx1
assign ALUOp[0] = i_add | i_lw | i_sw | i_addi | i_and | i_slt |
i_addu | i_nor | i_slti | i_andi | i_srl | i_sllv;

//这些操作对应的ALU操作码都是xx1x
assign ALUOp[1] = i_sub | i_beq | i_and | i_sltu | i_subu | i_lui |
i_bne | i_andi | i_srl | i_sllv;

//这些操作对应的ALU操作码都是x1xx
assign ALUOp[2] = i_or | i_ori | i_slt | i_sltu | i_slti | i_sllv |
i_srlv;

//这些操作对应的ALU操作码都是1xxx
assign ALUOp[3] = i_sll | i_nor | i_lui | i_srl | i_srlv;
```

依次执行这四行，得到的ALUOp便指示了ALU的操作

2.8. 功能6：写回时的写使能

2.8.1. RegWrite：寄存器写使能



1️⃣RegWrite信号：

1. 控制是否将操作的结果写回到寄存器中(RegWrite=1写回)
2. 该信号的设置取决于当前执行的指令类型

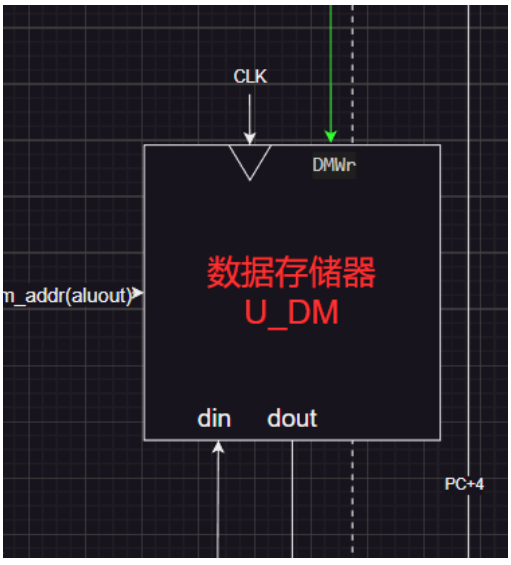
2️⃣以下两种情况才需要写回

1. 指令为R型(jr除外)，
2. 指令为lw/addi/iri/jar/lui/slti/andi/jalr时，需要写回给寄存器

3️⃣代码实现

```
RegWrite = rtype & ~i_jr | i_lw | i_addi | i_ori | i_jal | i_lui |
i_slti | i_andi | i_jalr;
```

2.8.2. MemWrite：内存写使能



1️⃣MemWrite信号：

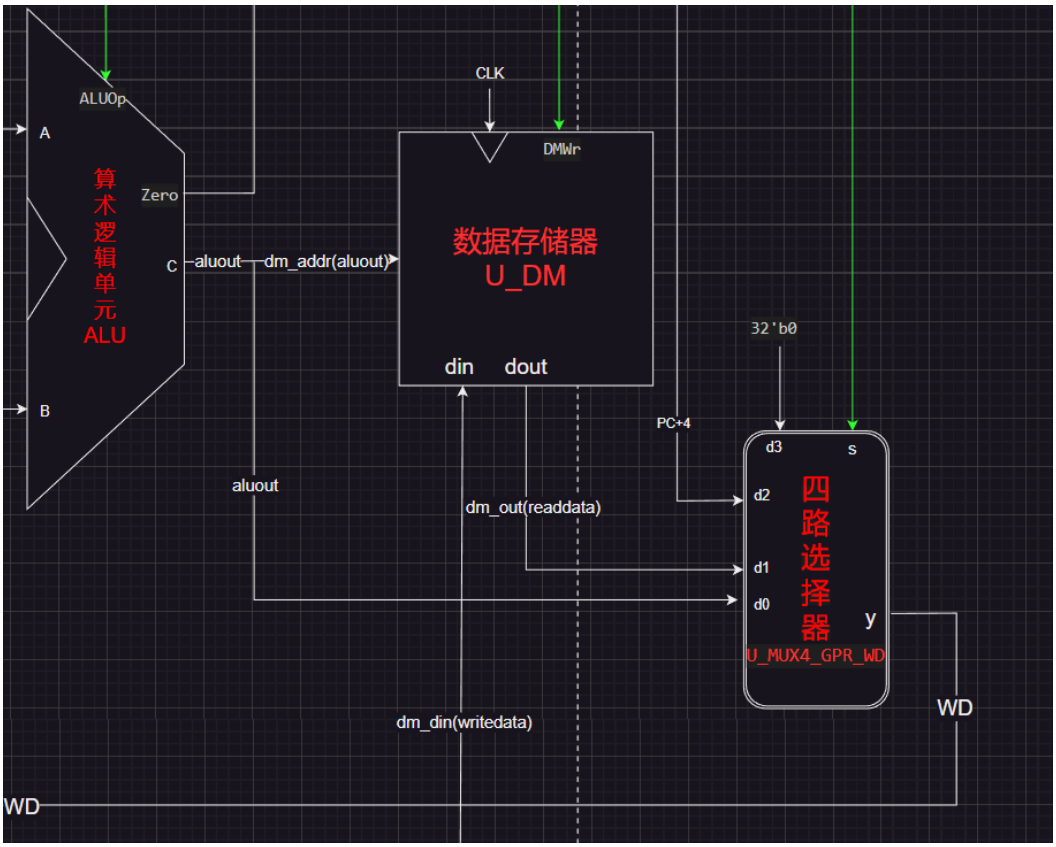
1. 为1时结果可以写回到内存，主要用于执行特定需要写内存指令时，激活内存写
2. MIPS中其实也只有sw指令需要执行内存操作，用于将数据从寄存器写入到内存中的指定地址

2️⃣代码实现：

```
MemWrite = i_sw;
```

2.9. 功能7：数据写回写什么/写到哪

2.9.1. WDSel：要将什么数据写回？(图右下角)



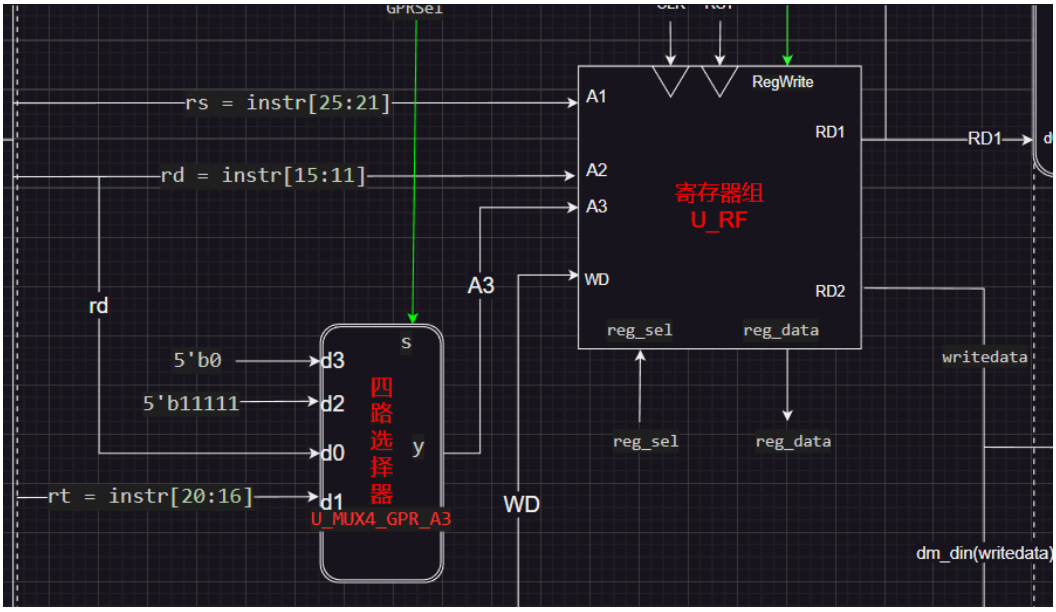
1️⃣四路选择机制

条件	含义
WDSel=00	数据来自 ALU 的运算结果
WDSel=01	数据来自内存，对应的指令有且只有内存加载lw
WDSel=10	数据来自程序计数器，通常用于跳转并链接指令
WDSel=11	操作非法

2️⃣代码实现：首先默认是有WDSel=00的

```
//当执行lw指令，会有WDSel=01，意味着写回阶段应该从内存读取数据并写入指定寄存器
assign WDSel[0] = i_lw;
//当执行跳转指令时，会有WDSel=10，指示写回阶段的数据应该来自PC
assign WDSel[1] = i_jal | i_jalr;
```

2.9.2. GPRSel：要写回到那个通用寄存器？（左下角）



1️⃣关于RD和RT寄存器

1. 在R型指令中，`RD` 字段指定了运算结果应该被写入的寄存器
2. 在I型指令中，`RT` 字段指定了运算结果应该被写入的寄存器

2️⃣四路选择机制

条件	含义
GPRSel=00	对应R型指令，将运算结果送给由指令中的 `RD` 字段指定的寄存器
GPRSel=01	对应I型指令，将运算结果送给由指令中的 `RT` 字段指定的寄存器
GPRSel=10	对应jal命令，执行该指令时，结果统一写入31号寄存器
GPRSel=11	操作非法

3️⃣代码实现：首先默认是有GPRSel=00的

```
//当执行lw, addi, ori, lui, slti, andi指令之一时，会有GPRSel=01，对应i指令
assign GPRSel[0] = i_lw | i_addi | i_ori | i_lui | i_slti | i_andi;
//当执行jal指令之一时，会有GPRSel=10，对应jal指令
assign GPRSel[1] = i_jal;
```

2.10. 总体代码`alu.v`

```
`include "ctrl_encode_def.v"

module ctrl(Op, Funct, Zero,
            RegWrite, MemWrite,
            EXTOp, ALUOp, NPCOp,
            ALUSrc, GPRSel, WDSel, ALUSrcA
            );

input  [5:0] Op;       // opcode
```

```
input  [5:0] Funct;    // funct
input        Zero;

output       RegWrite; // control signal for register write
output       MemWrite; // control signal for memory write
output       EXTOp;    // control signal to signed extension
output [3:0] ALUOp;    // ALU opertion
output [1:0] NPCOp;    // next pc operation
output       ALUSrc;   // ALU source for B  0:B来自RD2; 1:B来自立即数
output       ALUSrcA;  // ALU source for A  0:A来自RD1;   1:A来自shamt
output [1:0] GPRSel;   // general purpose register selection 对应A3选择器
output [1:0] WDSel;    // (register) write data selection 对应连接WD的选择器

//R型指令
wire rtype  = ~|Op;
wire i_add  = rtype&Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; // add
wire i_sub  = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&Funct[1]&~Funct[0]; // sub
wire i_and  = rtype& Funct[5]&~Funct[4]&~Funct[3]&Funct[2]&~Funct[1]&~Funct[0]; // and
wire i_or   = rtype& Funct[5]&~Funct[4]&~Funct[3]&Funct[2]&~Funct[1]& Funct[0]; // or
wire i_slt  = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&Funct[1]&~Funct[0]; // slt
wire i_sltu = rtype& Funct[5]&~Funct[4]& Funct[3]&~Funct[2]&Funct[1]& Funct[0]; // sltu
wire i_addu = rtype&Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]& Funct[0]; // addu
wire i_subu = rtype& Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&Funct[1]& Funct[0]; // subu
wire i_sll  = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; //sll
wire i_nor  = rtype& Funct[5]&~Funct[4]&~Funct[3]& Funct[2]&Funct[1]& Funct[0]; //nor
wire i_srl  = rtype&~Funct[5]&~Funct[4]&~Funct[3]&~Funct[2]&Funct[1]&~Funct[0]; //srl
wire i_sllv = rtype&~Funct[5]&~Funct[4]&~Funct[3]&Funct[2]&~Funct[1]&~Funct[0];  //sllv
wire i_srlv = rtype&~Funct[5]&~Funct[4]&~Funct[3]&Funct[2]&Funct[1]&~Funct[0];  //srlv

//I型指令
wire i_addi = ~Op[5]&~Op[4]& Op[3]&~Op[2]&~Op[1]&~Op[0]; // addi
wire i_ori  = ~Op[5]&~Op[4]& Op[3]& Op[2]&~Op[1]& Op[0]; // ori
wire i_lw   =  Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0]; // lw
wire i_sw   =  Op[5]&~Op[4]& Op[3]&~Op[2]& Op[1]& Op[0]; // sw
wire i_beq  = ~Op[5]&~Op[4]&~Op[3]& Op[2]&~Op[1]&~Op[0]; // beq
wire i_lui  = ~Op[5]&~Op[4]& Op[3]& Op[2]& Op[1]& Op[0]; // lui
wire i_slti = ~Op[5]&~Op[4]& Op[3]& ~Op[2]& Op[1]& ~Op[0]; // slti
wire i_bne  = ~Op[5]&~Op[4]& ~Op[3]& Op[2]& ~Op[1]& Op[0]; // bne
wire i_andi = ~Op[5]&~Op[4]& Op[3]& Op[2]& ~Op[1]& ~Op[0]; //andi

//J型指令
```

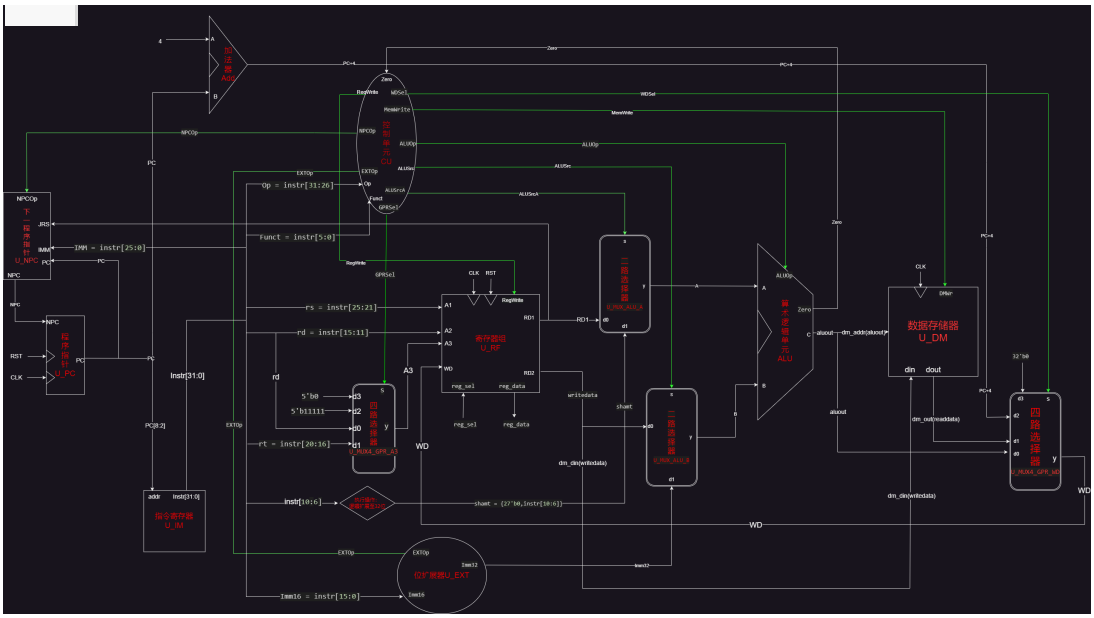
```
wire i_j    = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]&~Op[0];  // j
wire i_jal  = ~Op[5]&~Op[4]&~Op[3]&~Op[2]& Op[1]& Op[0];  // jal
wire i_jr   = rtype&
~Funct[5]&~Funct[4]&Funct[3]&~Funct[2]&~Funct[1]&~Funct[0]; // jr op
段与r型指令一样全为0
wire i_jalr = rtype&
~Funct[5]&~Funct[4]&Funct[3]&~Funct[2]&~Funct[1]&Funct[0]; // jalr

//结果是否要写入寄存器
assign RegWrite   = rtype&~i_jr | i_lw | i_addi | i_ori | i_jal |
i_lui | i_slti | i_andi | i_jalr; // register write

//结果是否要写入内存
assign MemWrite   = i_sw;

//ALU的A两端输入
assign ALUSrcA  = i_sll | i_srl;
assign ALUSrc     = i_lw | i_sw | i_addi | i_ori | i_lui | i_slti |
i_andi;

//是否要进行符号扩展
assign EXTOp      = i_addi | i_lw | i_sw | i_lui | i_slti;


//要写回到那个通用寄存器
assign GPRSel[0] = i_lw | i_addi | i_ori | i_lui | i_slti | i_andi;
assign GPRSel[1] = i_jal;

//要将什么数据写回
assign WDSel[0] = i_lw;
assign WDSel[1] = i_jal | i_jalr;

//NPC
assign NPCOp[0] = i_beq & Zero | i_bne & ~Zero | i_jr | i_jalr;
assign NPCOp[1] = i_j | i_jal | i_jr | i_jalr;

//ALU
assign ALUOp[0] = i_add | i_lw | i_sw | i_addi | i_and | i_slt |
i_addu | i_nor | i_slti | i_andi | i_srl | i_sllv;
assign ALUOp[1] = i_sub | i_beq | i_and | i_sltu | i_subu | i_lui |
i_bne | i_andi | i_srl | i_sllv;
assign ALUOp[2] = i_or | i_ori | i_slt | i_sltu | i_slti | i_sllv |
i_srlv;
assign ALUOp[3] = i_sll | i_nor | i_lui | i_srl | i_srlv;

endmodule
```

2. 单周期CPU模块：`SCCPU.v`

2.1. 概述

1. 目的：把除了存储器在内的各个模块，协调在一起

2. 实例化后的数据通路(重点重点重点)：想要看放大版的，访问URL[单周期CPU数据通路]



2.2. 实现的功能

1️⃣获取指令：使用程序计数器 `PC` 来获取当前指令`instr`

2️⃣指令译码

1. 将获取的指令`instr`分割为多部份，以适应指令各个模块的实际意义
2. 将16位的立即数扩展为32位

3️⃣生成控制信号：控制单元根据分割得到的操作码/功能码，生成控制信号，如ALUOp等

4️⃣执行：ALU根据控制单元生成的控制信号，执行相应的运算

5️⃣内存访问：(这点要在连接存储器后实现，也就是后边)如果执行的是lw或者sw指令，则可能还会将ALU运算结果写入/读出内存

6️⃣写回：根据WDSel和GPRSel信号，将选中的数据，写回到寄存器组中指定的寄存器中

7️⃣更新PC：根据执行的指令是否为跳转或分支指令以及是否满足跳转条件来更新 `PC`

2.3. 代码实现

```
module sccpu( clk, rst, instr, readdata, PC, MemWrite, aluout,
writedata, reg_sel, reg_data);

   input      clk;          // clock
   input      rst;          // reset

   input [31:0]  instr;     // instruction
   input [31:0]  readdata;  // data from data memory

   output [31:0] PC;        // PC address
   output        MemWrite;  // memory write
   output [31:0] aluout;    // ALU output
   output [31:0] writedata; // data to data memory

   input  [4:0] reg_sel;    // register selection (for debug use)
   output [31:0] reg_data;  // selected register data (for debug
use)
```

```
16
17    wire        RegWrite;    // control signal to register write
18    wire        EXTOp;       // control signal to signed extension
19    wire [3:0]  ALUOp;       // ALU opertion
20    wire [1:0]  NPCOp;       // next PC operation
21
22    wire [1:0]  WDSel;       // (register) write data selection
23    wire [1:0]  GPRSel;      // general purpose register selection
24
25    wire        ALUSrcA;     // ALU source for A    0:A来自RD1; 1:A来
自shamt
26    wire        ALUSrc;      // ALU source for B    0:B来自RD2; 1:B来
自立即数
27    wire        Zero;        // ALU ouput zero
28
29    wire [31:0] NPC;         // next PC
30
31    wire [4:0]  rs;          // rs
32    wire [4:0]  rt;          // rt
33    wire [4:0]  rd;          // rd
34    wire [5:0]  Op;          // opcode
35    wire [5:0]  Funct;       // funct
36    wire [15:0] Imm16;       // 16-bit immediate
37    wire [31:0] Imm32;       // 32-bit immediate
38    wire [25:0] IMM;         // 26-bit immediate (address)
39    wire [4:0]  A3;          // register address for write
40    wire [31:0] WD;          // register write data
41    wire [31:0] RD1;         // register data specified by rs
42    wire [31:0] B;           // operator for ALU B
43    wire [31:0] A;           // operator for ALU A
44    wire [31:0] shamt;       //偏移量
45
46    assign Op = instr[31:26];  // instruction
47    assign Funct = instr[5:0]; // funct
48    assign rs = instr[25:21];  // rs
49    assign rt = instr[20:16];  // rt
50    assign rd = instr[15:11];  // rd
51    assign Imm16 = instr[15:0];// 16-bit immediate
52    assign IMM = instr[25:0];  // 26-bit immediate
53    assign shamt = {27'b0,instr[10:6]}; // 5-bit 偏移量
54
55    // instantiation of control unit
56    ctrl U_CTRL (
57        .Op(Op), .Funct(Funct), .Zero(Zero),
58        .RegWrite(RegWrite), .MemWrite(MemWrite),
59        .EXTOp(EXTOp), .ALUOp(ALUOp), .NPCOp(NPCOp),
60        .ALUSrc(ALUSrc), .GPRSel(GPRSel), .WDSel(WDSel),
.ALUSrcA(ALUSrcA)
61    );
62
63    // instantiation of PC
64    PC U_PC (
65        .clk(clk), .rst(rst), .NPC(NPC), .PC(PC)
66    );
67
68    // instantiation of NPC
```

```
    NPC U_NPC (
        .PC(PC), .NPCOp(NPCOp), .IMM(IMM), .JRS(RD1), .NPC(NPC)
    );

    // instantiation of register file
    RF U_RF (
        .clk(clk), .rst(rst), .RFWr(RegWrite),
        .A1(rs), .A2(rt), .A3(A3),
        .WD(WD),
        .RD1(RD1), .RD2(writedata),
        .reg_sel(reg_sel),
        .reg_data(reg_data)
    );

    // mux for register data to write
    mux4 #(5) U_MUX4_GPR_A3 (
        .d0(rd), .d1(rt), .d2(5'b11111), .d3(5'b0), .s(GPRSel), .y(A3)
//d2对应第31号寄存器
    );

    // mux for register address to write
    mux4 #(32) U_MUX4_GPR_WD (
        .d0(aluout), .d1(readdata), .d2(PC + 4), .d3(32'b0),
.s(WDSel), .y(WD)
    );

    // mux for signed extension or zero extension
    EXT U_EXT (
        .EXTOp(Imm16), .EXTOp(EXTOp), .Imm32(Imm32)
    );

    mux2 #(32) U_MUX_ALU_A (
          .d0(RD1), .d1(shamt), .s(ALUSrcA), .y(A)
    );
    // mux for ALU B
    mux2 #(32) U_MUX_ALU_B (
        .d0(writedata), .d1(Imm32), .s(ALUSrc), .y(B)
    );

    // instantiation of alu
    alu U_ALU (
        .A(A), .B(B), .ALUOp(ALUOp), .C(aluout), .Zero(Zero)
    );

endmodule
```

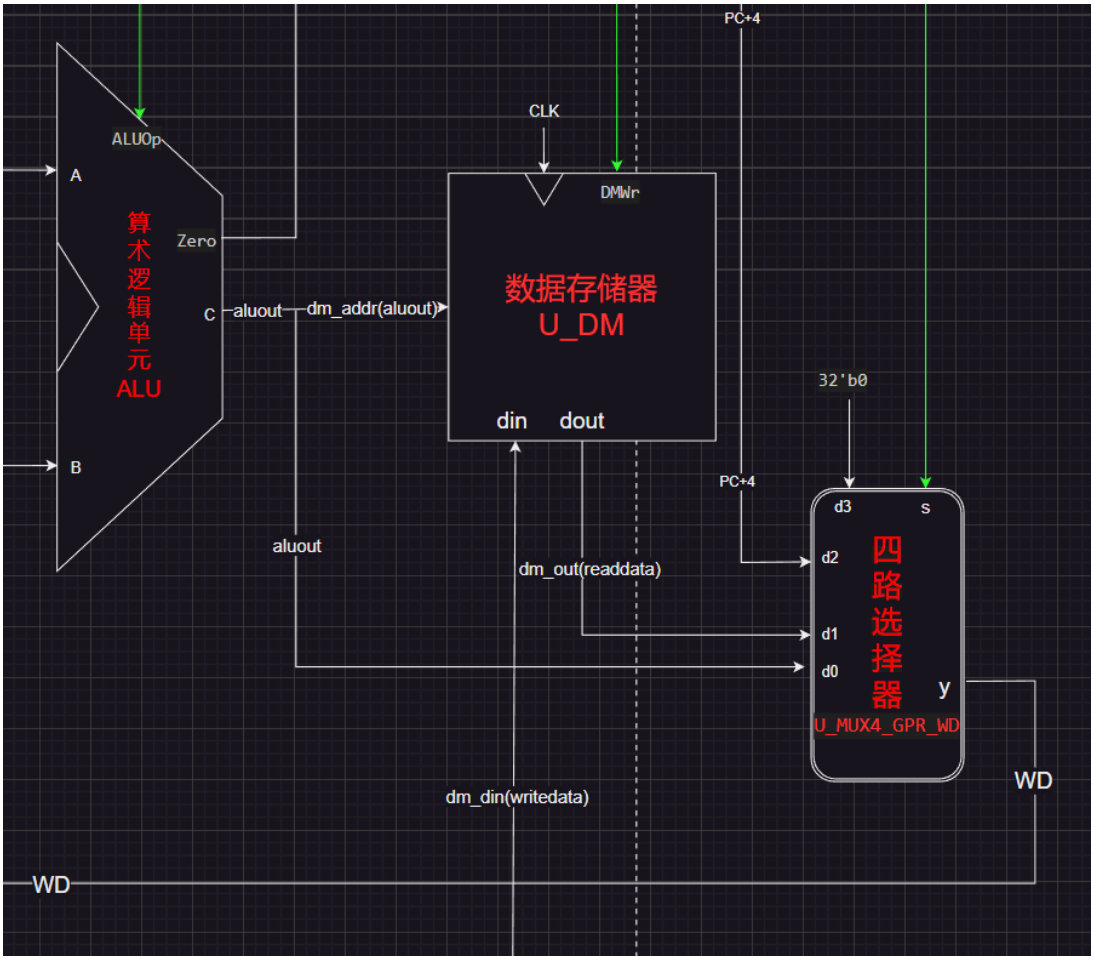
3. 连接CPU和内存

3.1. 源代码

```
module sccomp(clk, rstn, reg_sel, reg_data);
   input          clk;
   input          rstn;
   input [4:0]    reg_sel;
```

```
    output [31:0]  reg_data;

    wire [31:0]    instr;
    wire [31:0]    PC;
    wire           MemWrite;
    wire [31:0]    dm_addr, dm_din, dm_dout;

    wire rst = ~rstn;

  // instantiation of single-cycle CPU
    sccpu U_SCCPU(
          .clk(clk),                // input:  cpu clock
          .rst(rst),                // input:  reset
          .instr(instr),            // input:  instruction
          .readdata(dm_dout),       // input:  data to cpu
          .MemWrite(MemWrite),      // output: memory write signal
          .PC(PC),                  // output: PC
          .aluout(dm_addr),         // output: address from cpu to
memory
          .writedata(dm_din),       // output: data from cpu to
memory
          .reg_sel(reg_sel),        // input:  register selection
          .reg_data(reg_data)       // output: register data
          );

  // instantiation of data memory
    dm    U_DM(
          .clk(clk),           // input:  cpu clock
          .DMWr(MemWrite),     // input:  ram write
          .addr(dm_addr[8:2]), // input:  ram address
          .din(dm_din),        // input:  data to ram
          .dout(dm_dout)       // output: data from ram
          );

  // instantiation of intruction memory (used for simulation)
    im    U_IM (
       .addr(PC[8:2]),     // input:  rom address
       .dout(instr)        // output: instruction
    );

endmodule
```

3.2. 注意事项



注意这几行代码

```
.readdata(dm_dout),
.aluout(dm_addr),
.writedata(dm_din),
```

CPU作为模块被实例化后，这几个端口的名称发生了变化，图中已用括号标出，详见上图