5 单周期MIPS CPU设计 SC CPU

英语

1. *periodic*周期的
2. *instantaneously*即刻地
3. *schematic*示意图
1. *oscillates*摆动
2. *so that*以便因此

📌**时钟周期定义为从一个时钟上升沿到下一个时钟的上升沿**

SC CPU每一个时钟周期执行一条指令完，时钟周期由**最复杂的指令执行时间**确定

在时钟的**上升沿**更新PC等寄存器状态

SC CPU是**最经济有效的但是是最耗时的**

> *A single-cycle CPU (central processing unit) executes each instruction in one clock cycle. CPU states (program counter (PC) and registers) are updated at the rising edges of the clock. The cycle time is determined so that the most complex instruction can complete the execution correctly in one clock cycle. Compared to other CPUs, the single-cycle CPU is the most cost-effective but time-consuming CPU.*

5.1执行一条指令所需要的电路

5.1.1取指阶段所需要的电路

CPU使用PC寄存器中的值作为指令存储器的地址，读出的内容即为要执行的指令，同时需要生成next_pc

在每个clk的上升沿，NPC→PC

> *The multiplexer (mux) in the figure is used to select the next PC, which will be written to the PC at the rising edge of the clock*

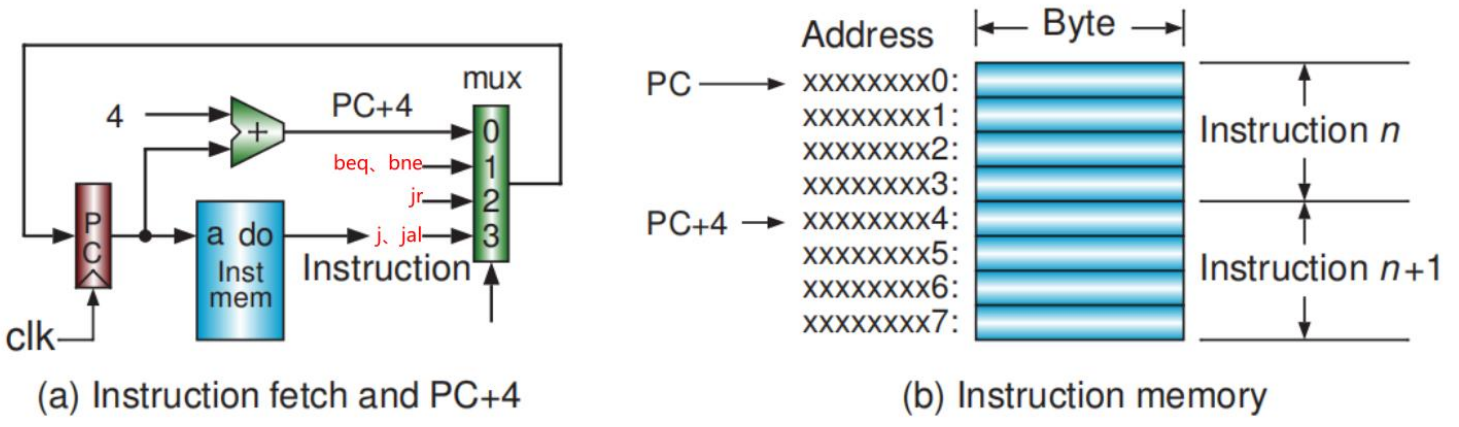


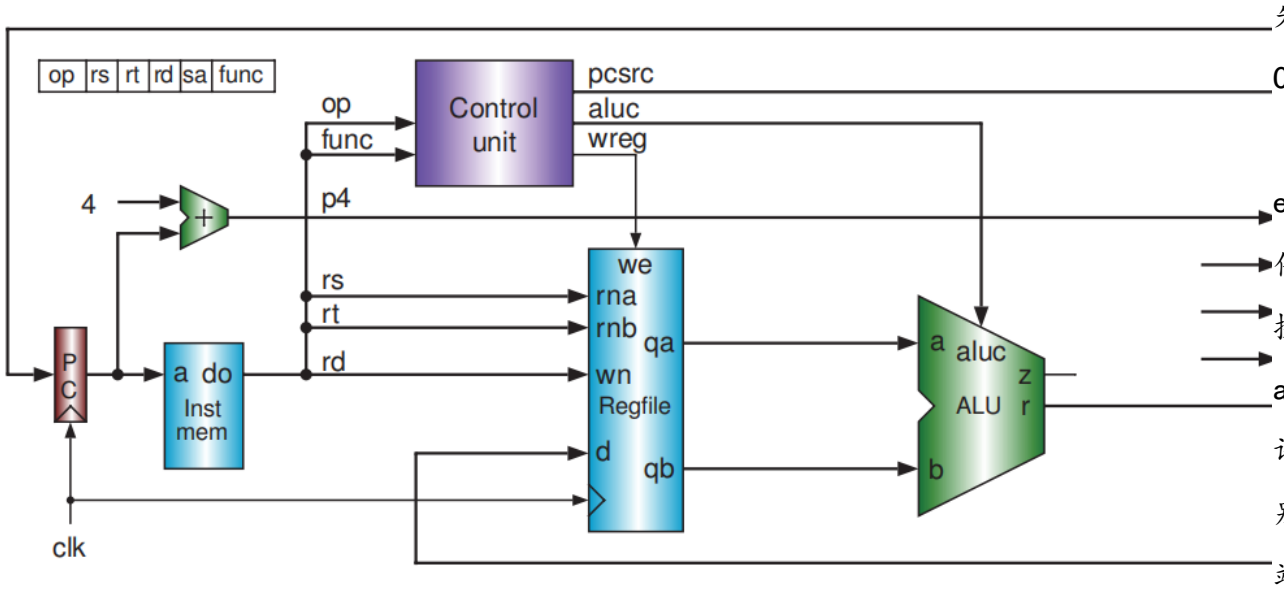
Figure 5.2 Block diagram of instruction fetch

5.1.2指令执行所需要的电路

执行一个指令所需要的部件包括：ALU、寄存器文件、控制单元CU

5.1.2.1 执行R指令所需要的电路

R指令一般是 助记符 rd,rs,rt➡对rs、rt执行对应的操作，然后写rd，此外还有移位指令 助记符 rd,rt,sa

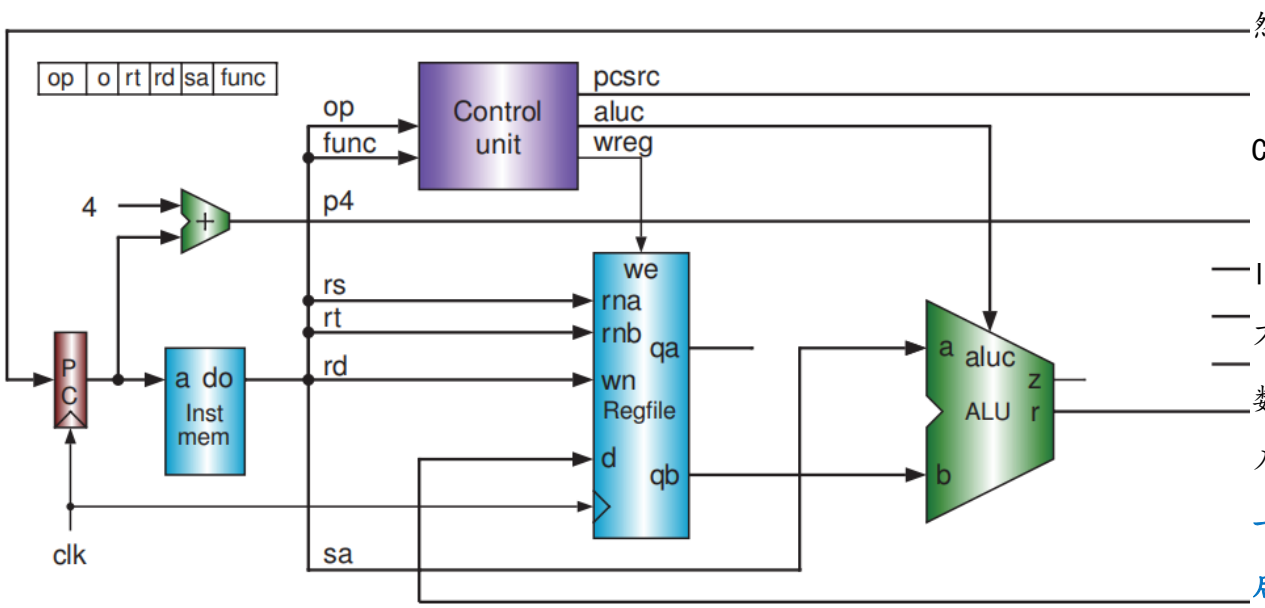


先考虑非移位指令，如左图：

CU译码

“aluc,wreg=1,pcsrc=0,sa_sel=0”给定rs、rt到寄存器文件，读出的qa、qb即为alu的操作数输入，根据CU的译码aluc执行对应的运算；根据CU译码的wreg信号，将rd和r分别输入到寄存器文件的wn、d端进行结果的写回。

Figure 5.3 Block diagram for R-format arithmetic and logic instructions



然后考虑移位指令，如右图

CU译码“aluc,wreg=1,pcsrc=0,sa_sel=1”。移位指令的ALU a端口不再使用ra,而是使用指令立即数imm[15:0]的[6:10]作为输入。因此**综合下来，a端口需要一个2选1多选器，sa=1时选择后者，sa=0时选qa**

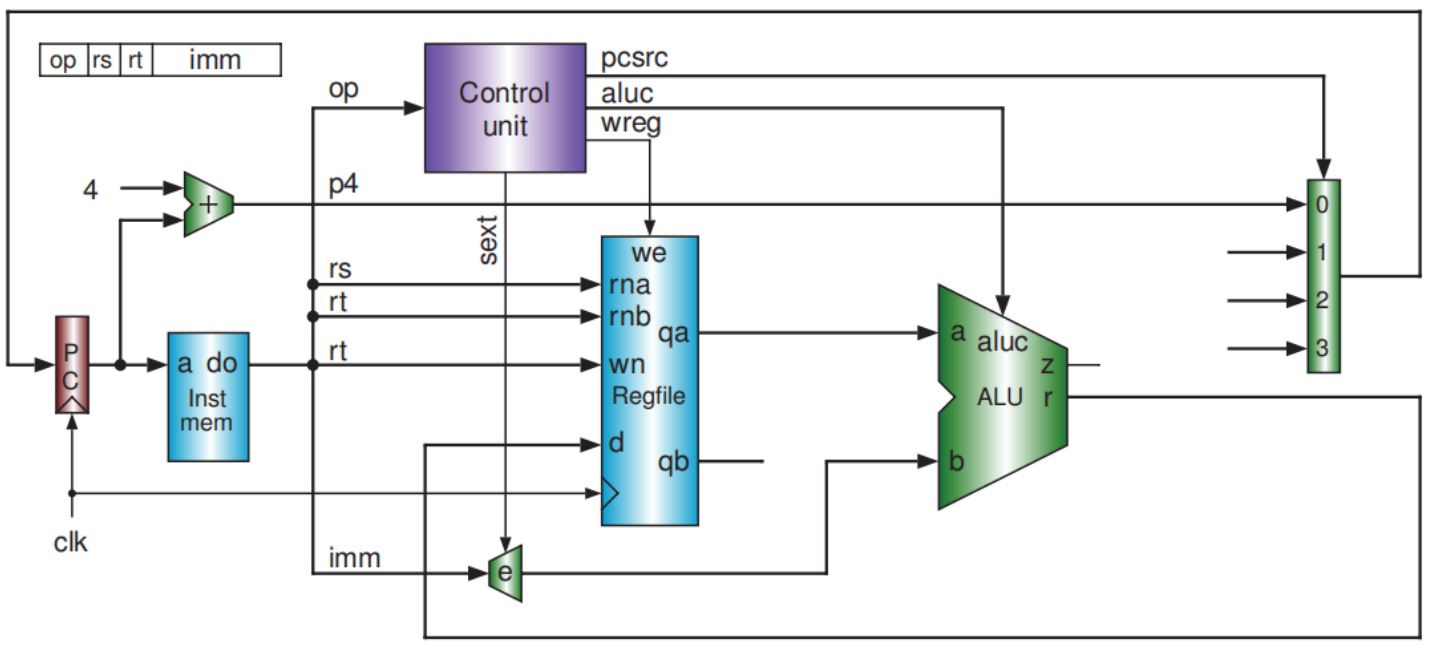
Figure 5.4 Block diagram for R-format shift instructions

5.1.2.2 执行I指令需要的电路

I指令包括立即数运算指令、load、store、分支指令等，下面分别进行介绍

指令格式为 助记符 rt,rs,立即数

立即数运算指令



ALU的a端输入即为rs,故sa_sel=0;

b端输入则不再是qb而是立即数的扩展——逻辑运算0扩展，其他符号扩展，因此**b端需要alusrc选**

Figure 5.5 Block diagram for I-format arithmetic and logic instructions

择信号，alusrc为1时选择扩展后的立即数，为0时选择qb；同时是否进行符号扩展也需要进行选择，CU译码输出控制信号sext作为是否符号扩展的标志

load/store指令

load/store与立即数运算指令不同的是“**store指令不写寄存器，且load指令写寄存器并不写alu运算数据**”

- store指令 sw rt,rs,立即数

  store指令将reg[rt]➡mem[reg[rs]+立即数]，多了一个写存阶段，需要再加一个存储器

  CU译码控制信号wmem=1,wreg=0,aluc=x000,sext=1,存储器的数据端口为qb,地址端口为alu输出r

- load指令lw,rt,rs,立即数

  load指令将mem[reg[rs]+立即数]➡reg[rt],比store多了一个写寄存器阶段

  CU译码控制信号wmem=0,wreg=1,aluc=x000,sext=1,存储器的输出数据端口接寄存器的数据写入端口,存储器的地址端口为alu输出r,寄存器的地址端口为rt

  因此**比之前提到的电路多了两个多路选择器——一个在写寄存器端口的选择,选择信号为regdst,regdst=1时选rt否则选rd；一个在寄存器输入数据端口,选择信号是mem2reg,memtoreg=1时选择存储器读出数据写回否则选择alu运算结果r**

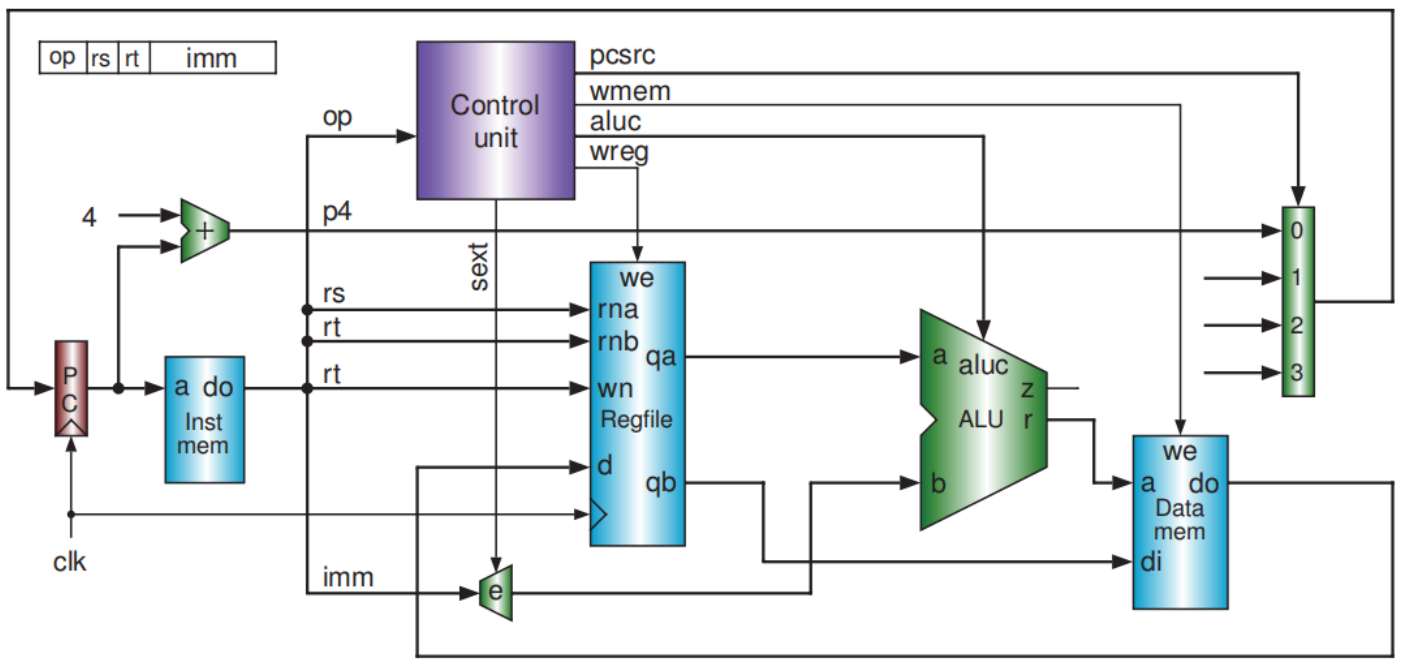


Figure 5.6 Block diagram for I-format load and store instructions

分支指令

分支指令是通过**比较rs和rt寄存器的值（异或累积取反非~|(reg[rs]^reg[rt]),值为1则相等，否则不等）**若满足助记符所表示的关系，那么就跳转到**立即数所形成的地址处（NPC=(PC+4)+(立即数符号扩展<<2)）**

CU根据指令op、funct译码出当前指令是分支指令branch=1，且**根据eq输入信号**，得到rs和rt寄存器的关系，若满足当前分支指令，则pc_src=01,否则pc_src=0

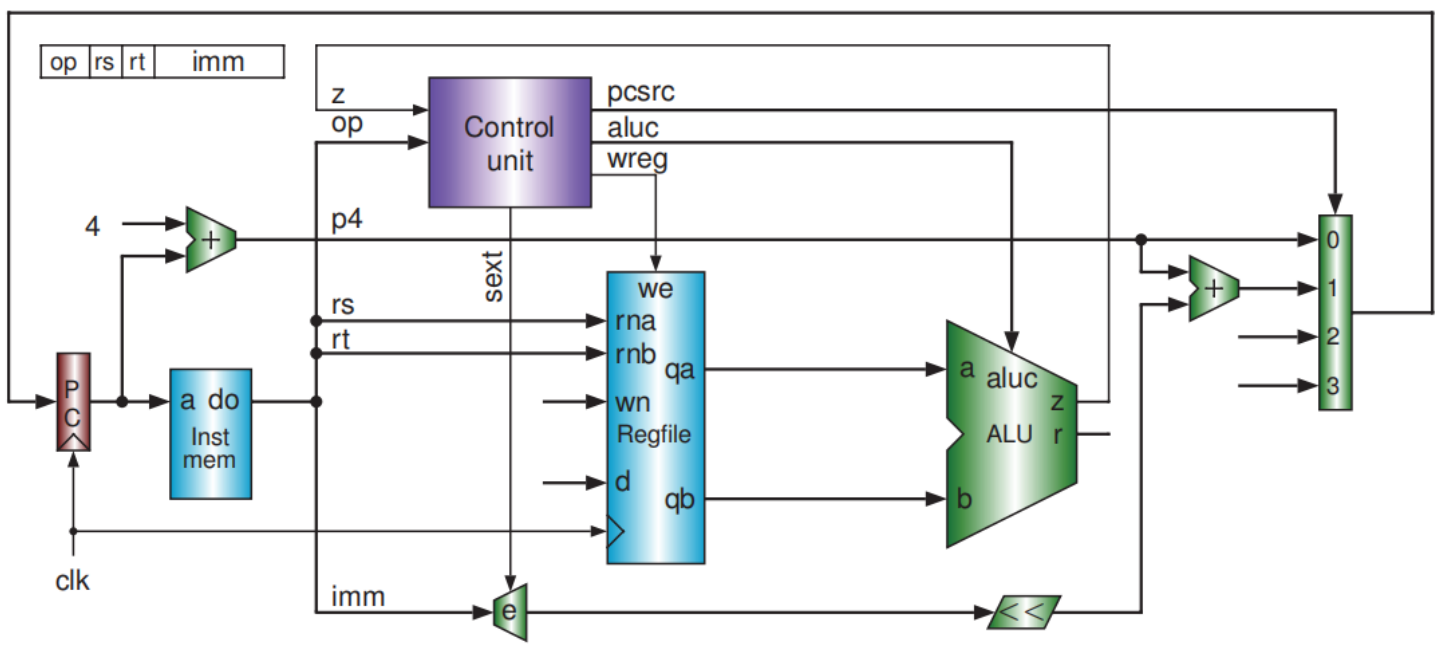


Figure 5.7 Block diagram for I-format branch instructions

5.1.2.3 J型指令所需要的电路

J型指令的格式是6位funct，26位立即数，包括j、jal、jr三种指令

j、jal

J、Jal指令的NPC计算方式相同➡{PC+4[31:28],inst[26:0],2'b0}

当CU根据op、funct译码出当前指令为j指令时，jump=1,wreg=0,wmem=0,pcsrc=11其余控制信号任意

当CU根据op、funct译码出当前指令为jal指令时，除j的功能外，还需将PC+4/PC+8写入$31,因此

jump=1,wreg=1,wmem=0,pcsrc=11,jal=1(用于写寄存器地址选择，如果jal=1那么地址为31,否则为regdst选出的地址)

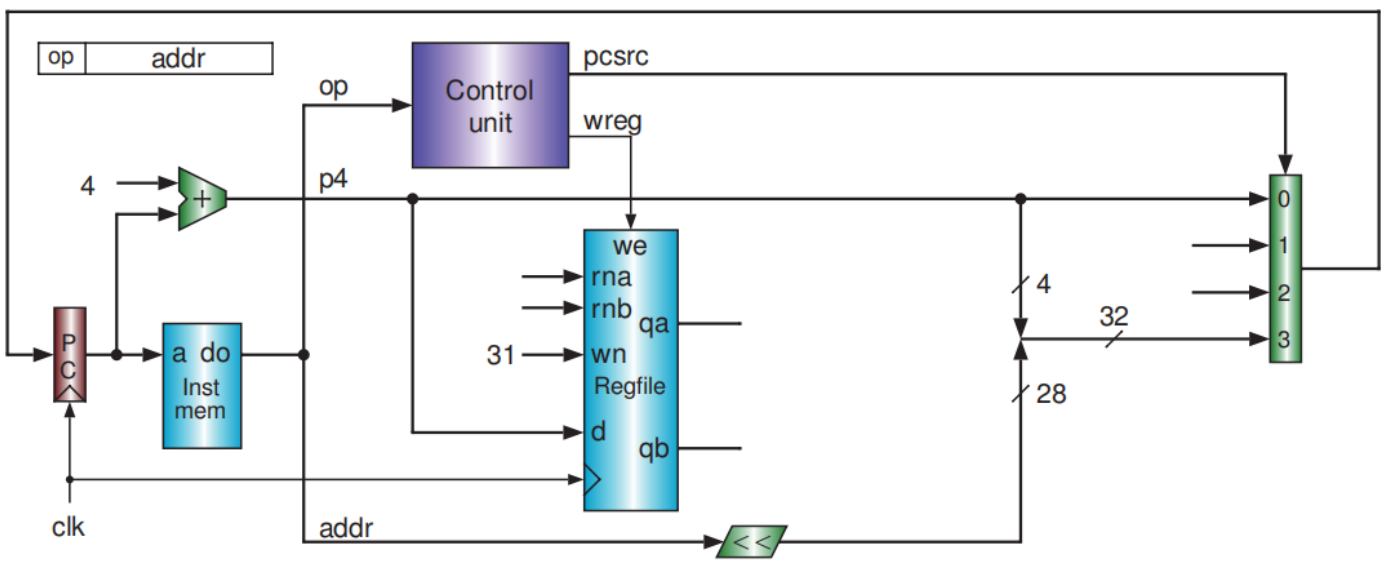


Figure 5.9 Block diagram for J-format jump-and-link instruction

jr指令

jr指令实质上属于R型指令：jr rd,rs,rt，功能是将reg[rs]作为NPC

CU译码当前指令为jr指令时,wmem=0,wreg=0,pcsrc=10,jump=1,jal=0

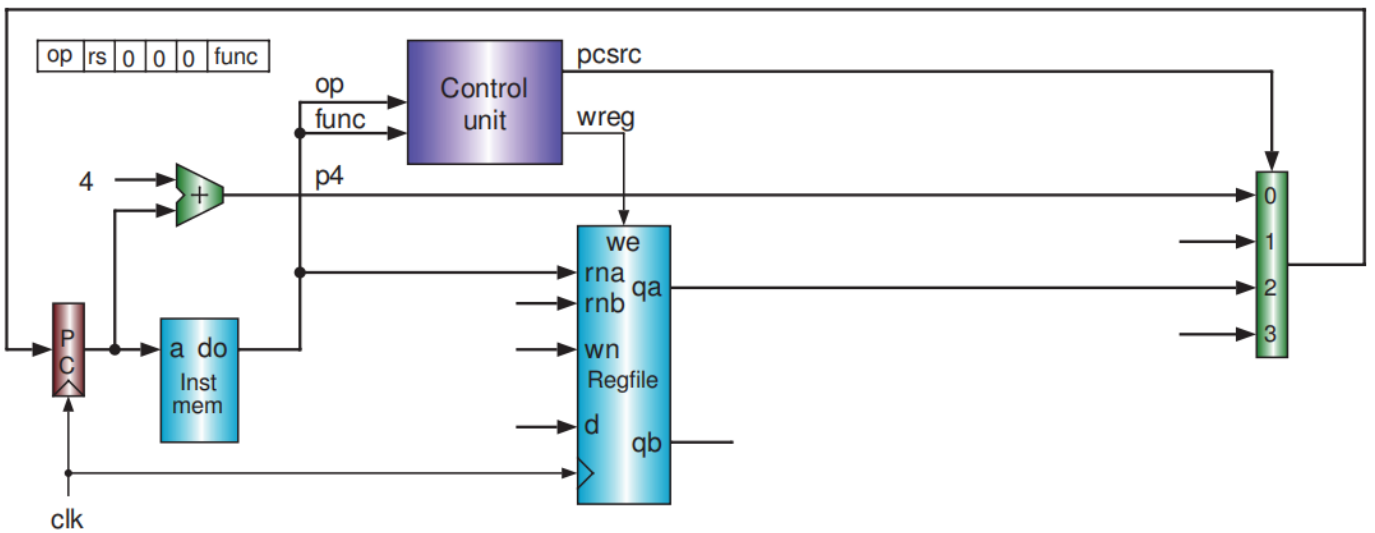


Figure 5.10 Block diagram for R-format jump-register instruction

5.2寄存器文件设计

根据5.1执行一条指令所需要的电路所涉及到的寄存器电路，一个寄存器文件应该需要包含32个寄存器、rna[4:0],rnb[4:0],wa[4:0],wd[31:0],we,clk,clrn,qa[31:0],qb[31:0]这些端口，其综合符号如下：

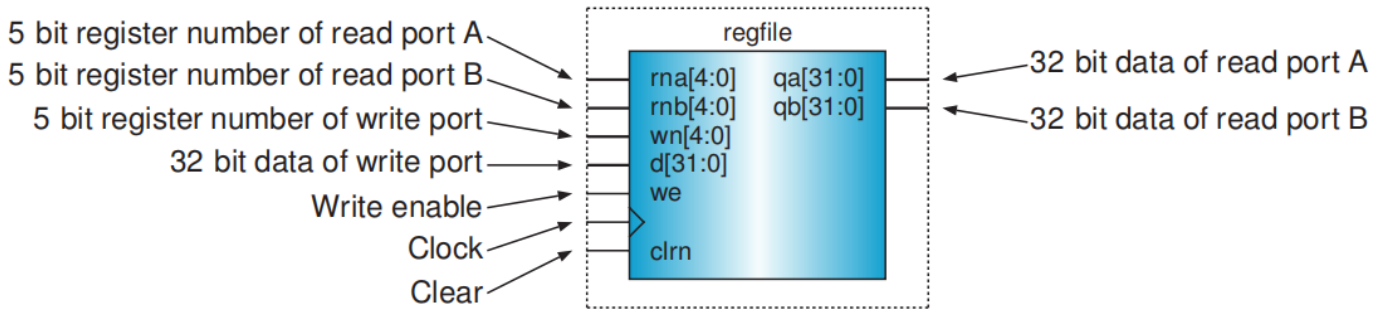


Figure 5.11 Symbol of the register file

这里为了简便，我们不采用构建触发器式寄存器的形式，而是直接使用位宽为32的reg数组实现寄存器文件，代码如下：

```
`timescale 1ns / 1ps

module regfile(
    input clk,we,clrn,

    input [4:0]rna,rnb,wn,
    input [31:0]wd,

    output [31:0]qa,qb
);
    reg [31:0]regs[31:0];
    always @(posedge clk or negedge clrn) begin
        if (clrn) begin:reg_clear
            integer i;
            for ( i= 0; i<32; i=i+1) begin
                regs[i]=32'b0;
            end
        end else if (we&&we!=0) begin
            regs[wn]<=wd;
        end
    end
    assign qa=(rna==5'b0)?0:regs[rna];
    assign qb=(rnb==5'b0)?0:regs[rnb];
endmodule
```

Verilog

5.3 SC CPU数据通路设计

📌 一个CPU是由datapath数据通路和control unit控制单元组成的

数据通路是执行数据处理操作的功能单元[注释1]的集合，而控制单元是用来管理数据通路的组件

5.3.1多路选择器的使用

5.3.3.1 选择产生NPC——选择信号pcsrc

SC CPU的PC寄存器在时钟上沿进行更新，需要在PC+4、分支指令的PC、J、Jal的PC以及Jr的r中四选1（如果是中断的话，4选一的结果还需要和epc的值二选一才能产生NPC）产生NPC

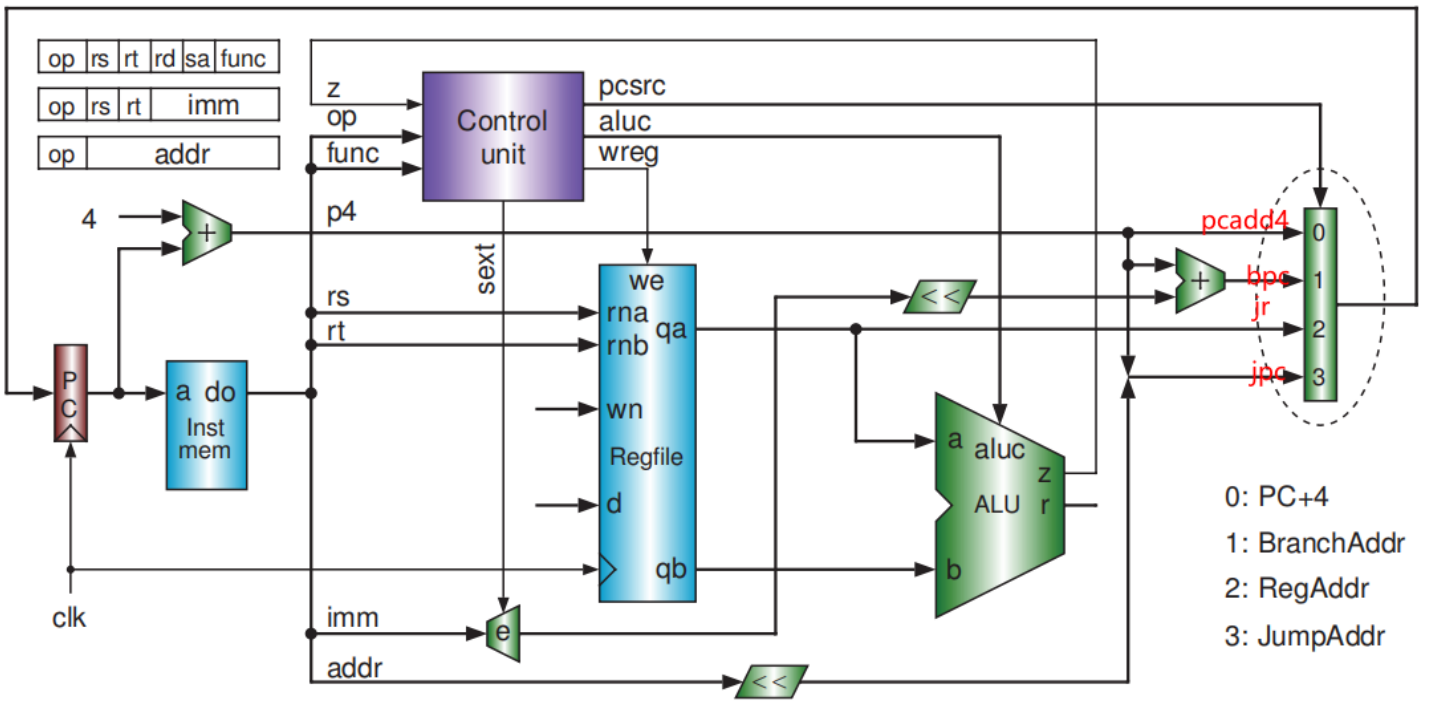


Figure 5.15 Using multiplexer for the next PC selection

5.3.3.2选择产生alu的运算数据——选择信号shift、alusrc

对于ALU的a操作数，输入可以是除移位指令外的寄存器数，也可以是移位指令的移位量，因此需要在这两个中二选一

→shift?imm[10:6]:qa

对于ALU的b操作数，输入可以是寄存器文件读出的qb，也可以是立即数扩展，因此需要在这两个中二选1→alusrc?立即数扩展:qb

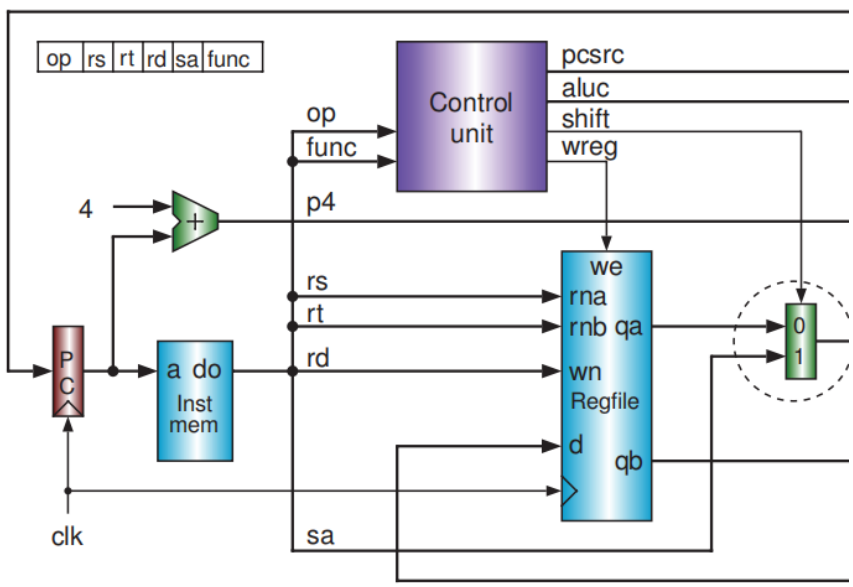


Figure 5.16 Using a multiplexer for ALU

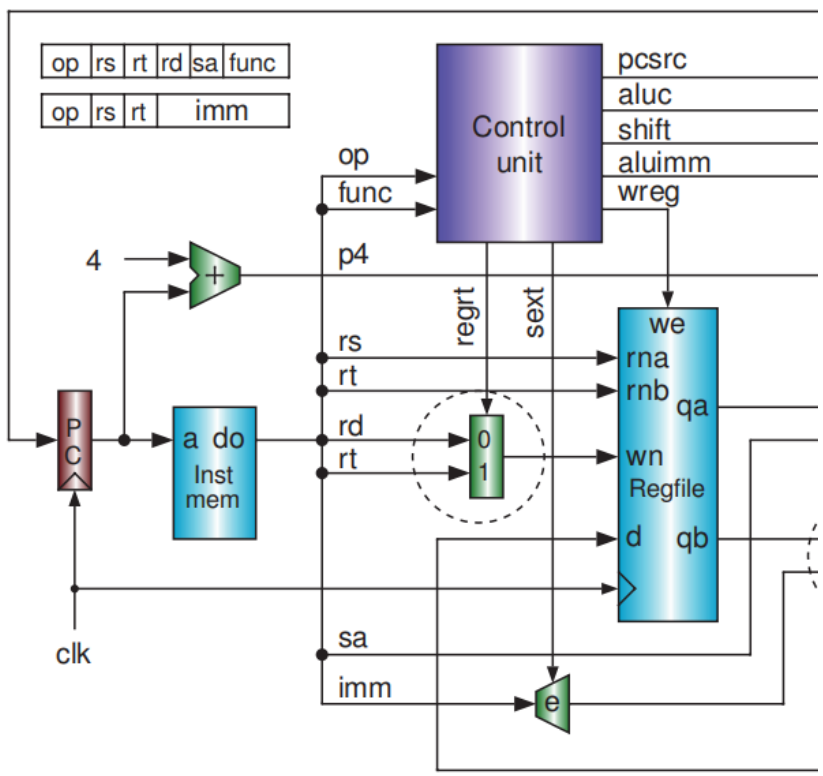


Figure 5.17 Using a multiplexer fo

5.3.3.3选择写寄存器的写入数据——m2reg

写寄存器可能是用ALU的运算结果(运算指令)也可能是存储器的读出结果(lw)也可能是pc+4/pc+8(jal)，用m2reg和jal信号选择——先m2reg选择aluout和memread然后再用jal选择m2reg的选择结果还是pc+4/+8

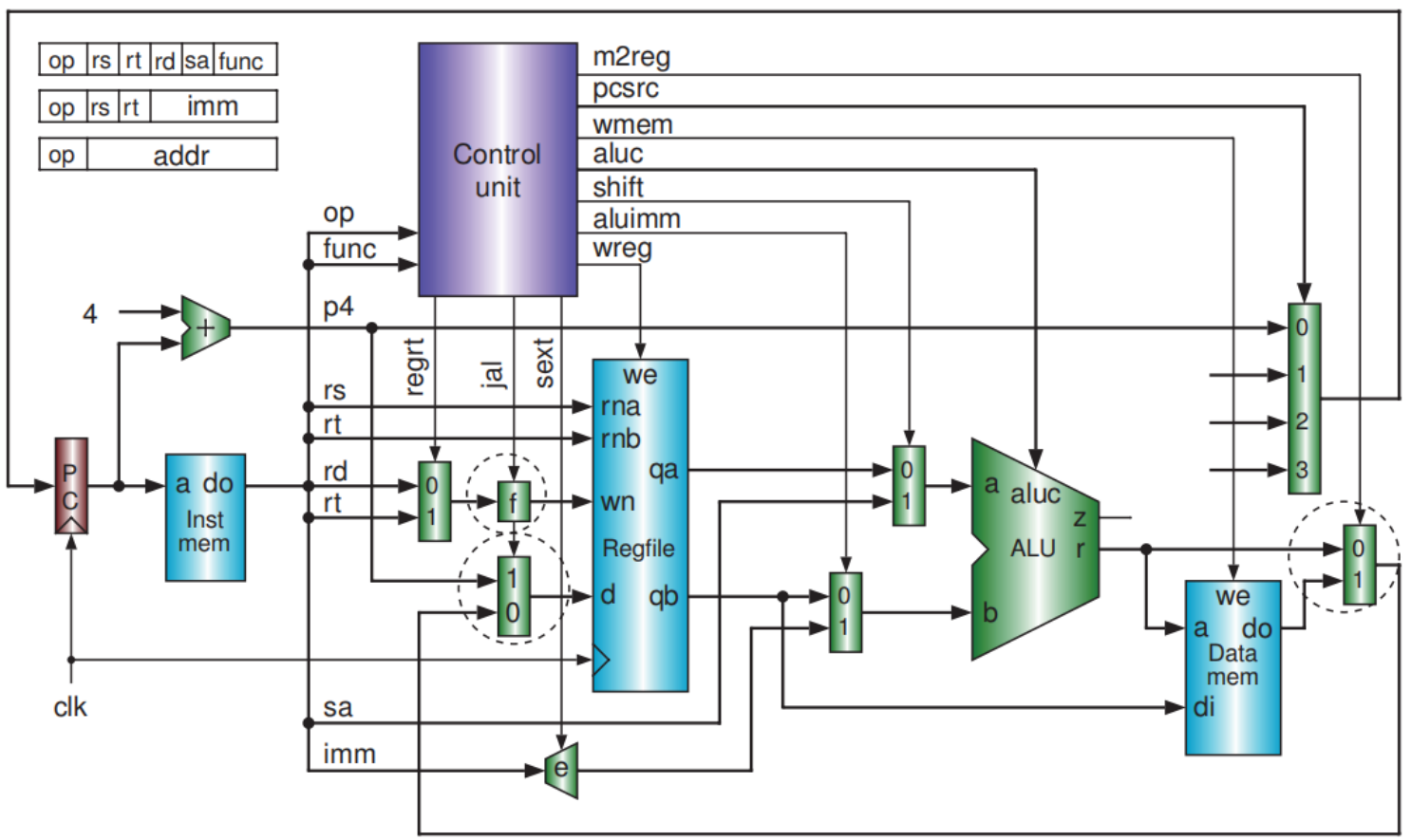


Figure 5.18 Using a multiplexer for register file written data selection

5.3.3.4写寄存器的寄存器编号——regdst、jal

写入的寄存器可以是rd，也可以是rt也可以是常量31，因此需要使用多选器来进行选择

rs、rt用选择信号regdst➡1为rt,0为rd

之后jal选择的结果可以用多选器也可以用表达式实现：选择结果|{5{jal}}

```
assign wn = reg_dest | {5{jal}};
```

5.3.3.4选择立即数扩展的类型——sext选择

逻辑运算时立即数是0扩展，其余指令均是符号扩展，因此需要进行选择，选择信号是sext

当sext为0时，执行无符号扩展，{16{sext},imm[15:0]};否则执行符号扩展{16{imm[15]},imm[15:0]}

因此可以写成表达式：{16{sext&imm[15]},imm[15:0]}

5.3.2 SC CPU综合示意图

通过5.1执行一条指令所需要的电路和5.3.1多路选择器的使用所分析得到的所有电路，我们可以综合得出SC、CPU的综合示意图如下：

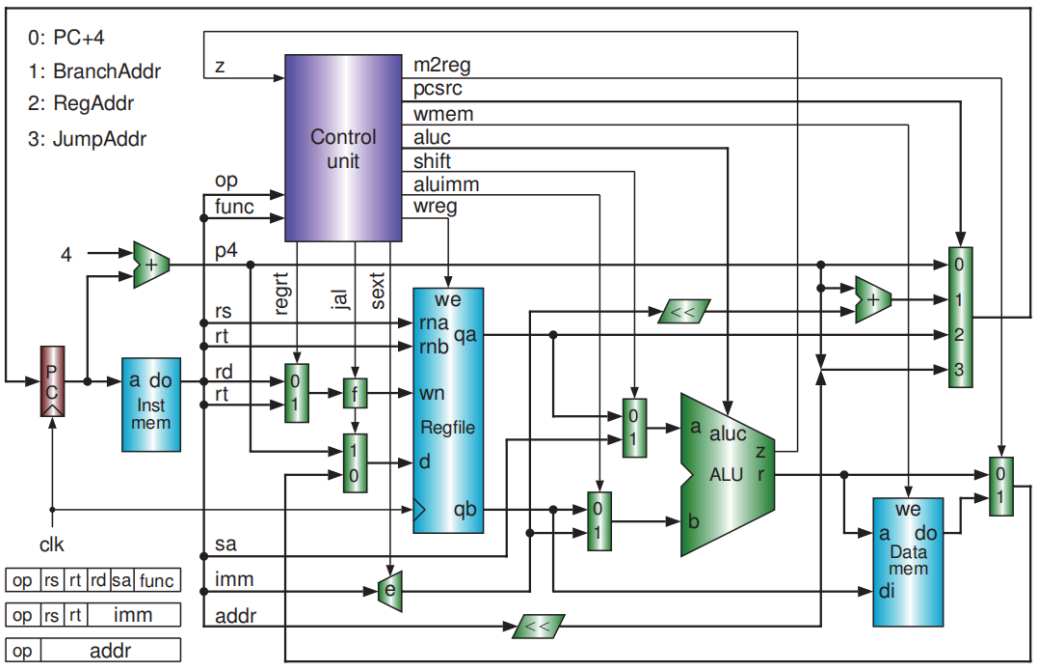


Figure 5.19 Block diagram of a single-cycle CPU

加上外部的存储器，得到SC 计算机的示意图如下：

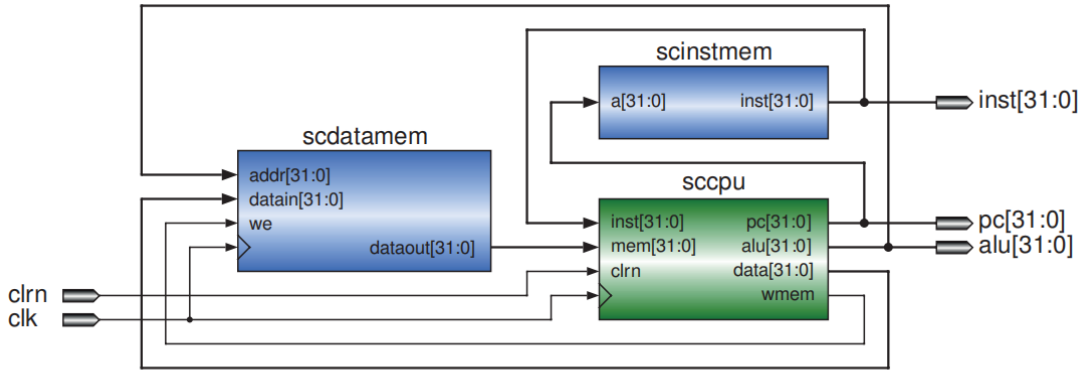


Figure 5.20 Block diagram of single-cycle computer

5.3.4 SC CPU数据通路实现

子模块

PC

```
`timescale 1ns / 1ps

module pc(
    input clk,clrn,
```

```
    input [31:0]npc,

    output reg[31:0]pc
);
    always @(posedge clk or posedge clrn) begin
        if (clrn) begin
            pc<=0;
        end else begin
            pc<=npc;
        end
    end
endmodule
```

Verilog

Adder

```
`timescale 1ns / 1ps

module adder(
    input [31:0]a,b,
    output [31:0]res
);
    assign res=a+b;
endmodule
```

Verilog

Mux2to1

```
`timescale 1ns / 1ps

module mux2to1(
    input [31:0]a,b,
    input sel,
```

```
    output [31:0]res
);
    assign res=sel?b:a;
endmodule
```

Verilog

Mux4to1

```
`timescale 1ns / 1ps

module mux4to1(
    input [31:0]a,b,c,d,
    input [1:0]sel,

    output reg[31:0]res
);
    always @(*) begin
        case (sel)
            2'b00:res=a;
            2'b01:res=b;
            2'b10:res=c;
            2'b11:res=d;
        endcase
    end
endmodule
```

Verilog

ALU

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
```

```
// Engineer:
//
// Create Date: 2023/09/02 14:37:56
// Design Name:
// Module Name: alu
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


`timescale 1ns / 1ps

module alu(
    input [31:0]a,b,//a端多路选择pc(pc的adder可以放在IF),sa,reg_a;b端多路选择
reg_b,imm,4(pc_adder),imm<<2;
    input [3:0]aluc,
    //x000 ADD,x100 SUB,x001 AND,x101 OR,x010 XOR,x110 LUI,0011 SLL,0111 SRL,1111 SRA
    output wire[31:0]res,
    output wire z
);
    reg [31:0]s0,s1,s2,s3;

    always @(*) begin
```

```
        casex (aluc)//带有x的匹配
            4'bx000:s0=a+b;
            4'bx100:s0=a+~b+1'b1;
            4'bx001:s1=a&b;
            4'bx101:s1=a|b;
            4'bx010:s2=a^b;
            4'bx110:s2={b[15:0],16'b0};
            4'b0011:s3=b<<a[4:0];
            4'b0111:s3=b>>a[4:0];
            4'b1111:s3=$signed(b)>>>a[4:0];
        endcase
    end
    assign z=(a==b)?1:0;
    mux4to1 mux4to1_inst(s0,s1,s2,s3,aluc[1:0],res);
endmodule
```

Verilog

RegFile

```
`timescale 1ns / 1ps

module regfile(
    input clk,we,clrn,

    input [4:0]rna,rnb,wn,
    input [31:0]wd,

    output [31:0]qa,qb
);
    reg [31:0]regs[31:0];
    always @(posedge clk or negedge clrn) begin
        if (clrn) begin:reg_clear
            integer i;
```

```Verilog
            for ( i= 0; i<32; i=i+1) begin
                regs[i]=32'b0;
            end
        end else if (we&&we!=0) begin
            regs[wn]<=wd;
        end
    end
    assign qa=(rna==5'b0)?0:regs[rna];
    assign qb=(rnb==5'b0)?0:regs[rnb];
endmodule
```

集成

```Verilog
`timescale 1ns / 1ps

module datapath(
    input clk,clrn,
    input regdst,alusrc,shift,m2reg,jal,
    input wreg,
    input sext,

    input [1:0]pcsrc,
    input [3:0]aluc,

    input [31:0]memread,
    input [31:0]inst,

    output eq,
    output [5:0]funct,op,
    output [31:0]pc,aluout,qb
);
    //pc赋值
```

```
    wire [31:0]npc,pcAdder4,bpc,jpc,rpc;
    mux4to1 npc_init(pcAdder4,bpc,rpc,jpc,pcsrc,npc);
    pc pc_init(clk,clrn,npc,pc);
    adder pc_adder(pc,32'd4,pcAdder4);
    assign jpc={pcAdder4[31:29],inst[25:0],2'b00};

    //赋值寄存器相关
    wire[4:0]rs,rt,rd;
    assign rs=inst[25:21];
    assign rt=inst[20:16];
    assign rd=inst[15:11];
    assign funct=inst[5:0];
    assign op=inst[31:26];

    //产生写寄存器地址
    wire [4:0]tempA,wn;
    mux2to1 mux2to1_wreg_A(rs,rt,regdst,tempA);
    assign wn=tempA|{5{jal}};

    //写寄存器数据选择
    wire [31:0]wregData_temp;
    mux2to1 generate_wregData_temp(aluout,memread,m2reg,wregData_temp);
    wire [31:0]wregData;
    mux2to1 generate_wregData(wregData_temp,pcAdder4,jal,wregData);
    //寄存器模块
    wire [31:0]qa;
    regfile regfile_init(clk,wreg,clrn,rs,rt,wn,wregData,qa,qb);
    assign rpc=qa;
    assign eq=~|(qa^qb);

    //立即数扩展模块
```

```verilog
    wire [15:0]imm=inst[15:0];
    wire [31:0]imm_sext={{16{sext&imm[15]}},imm};
    adder generate_bpc(pcAdder4,imm_sext<<2,bpc);

    //ALU模块
    wire [31:0]a,b;
    wire z;
    mux2to1 generate_alua(qa,imm[10:6],shift,a);
    mux2to1 generate_alub(qb,imm_sext,alusrc,b);
    alu alu_init(a,b,aluc,aluout,z);
endmodule
```

Verilog

5.4 SC CPU控制单元实现

指令操作译码

控制单元主要是根据输入的op、funct以及eq信号，产生各种控制信号输出

主要都是根据op来译码指令操作的，但是R型指令的op均为0，所有R型指令的操作译码还需要funct字段

下表即为根据op、funct使用全译码[注释2]方式实现对指令操作的译码——**为了不和指令助记符冲突，加i_前缀**

Table 5.1 Instruction decode

R-format			I- and J-format	
Inst.	op[5:0]	func[5:0]	Inst.	op[5:0]
i_add	000000	100000	i_addi	001000
i_sub	000000	100010	i_andi	001100
i_and	000000	100100	i_ori	001101
i_or	000000	100101	i_xori	001110
i_xor	000000	100110	i_lw	100011
i_sll	000000	000000	i_sw	101011
i_srl	000000	000010	i_beq	000100
i_sra	000000	000011	i_bne	000101
i_jr	000000	001000	i_lui	001111
			i_j	000010
			i_jal	000011

控制信号分类

控制信号可以分为四大类：多路选择器的选择信号、ALU的控制运算信号、寄存器和存储器的写使能信号、其他信号

1. 多选器的选择信号

   pcsrc、alusrc/aluimm、shift、m2reg、jal、regdst/regrt

2. ALU的控制运算信号：aluc

3. 寄存器和存储器的写使能信号：wreg、wmem

4. 其他：sext

Table 5.2 Control signals of single-cycle CPU

Signal	Meaning	Action
`wreg`	Write register	1: write; 0: do not write
`regrt`	Destination register is `rt`	1: select `rt`; 0: select `rd`
`jal`	Subroutine call	1: is `jal`; 0: is not `jal`
`m2reg`	Save memory data	1: select memory data; 0: select ALU result
`shift`	ALU A uses `sa`	1: select `sa`; 0: select register data
`aluimm`	ALU B uses immediate	1: select immediate; 0: select register data
`sext`	Immediate sign extend	1: sign-extend; 0: zero extend
`aluc[3:0]`	ALU operation control	x000: ADD; x100: SUB; x001: AND x101: OR; x010: XOR; x110: LUI 0011: SLL; 0111: SRL; 1111: SRA
`wmem`	Write memory	1: write memory; 0: do not write
`pcsrc[1:0]`	Next instruction address	00: select PC+4; 01: branch address 10: register data; 11: jump address

控制信号译码

根据之前译码分析得到的指令所要执行的操作，来分析所产生的控制信号，下图为控制信号的真值表

Table 5.3 Truth table of the control signals

Inst.	z	wreg	regrt	jal	m2reg	shift	aluimm	sext	aluc[3:0]	wmem	pcsrc[1:0]
i_add	x	1	0	0	0	0	0	x	x 0 0 0	0	0 0
i_sub	x	1	0	0	0	0	0	x	x 1 0 0	0	0 0
i_and	x	1	0	0	0	0	0	x	x 0 0 1	0	0 0
i_or	x	1	0	0	0	0	0	x	x 1 0 1	0	0 0
i_xor	x	1	0	0	0	0	0	x	x 0 1 0	0	0 0
i_sll	x	1	0	0	0	1	0	x	0 0 1 1	0	0 0
i_srl	x	1	0	0	0	1	0	x	0 1 1 1	0	0 0
i_sra	x	1	0	0	0	1	0	x	1 1 1 1	0	0 0
i_jr	x	0	x	x	x	x	x	x	x x x x	0	1 0
i_addi	x	1	1	0	0	0	1	1	x 0 0 0	0	0 0
i_andi	x	1	1	0	0	0	1	0	x 0 0 1	0	0 0
i_ori	x	1	1	0	0	0	1	0	x 1 0 1	0	0 0
i_xori	x	1	1	0	0	0	1	0	x 0 1 0	0	0 0
i_lw	x	1	1	0	1	0	1	1	x 0 0 0	0	0 0
i_sw	x	0	x	x	x	0	1	1	x 0 0 0	1	0 0
i_beq	0	0	x	x	x	0	0	1	x 0 1 0	0	0 0
i_beq	1	0	x	x	x	0	0	1	x 0 1 0	0	0 1
i_bne	0	0	x	x	x	0	0	1	x 0 1 0	0	0 1
i_bne	1	0	x	x	x	0	0	1	x 0 1 0	0	0 0
i_lui	x	1	1	0	0	x	1	x	x 1 1 0	0	0 0
i_j	x	0	x	x	x	x	x	x	x x x x	0	1 1
i_jal	x	1	x	1	x	x	x	x	x x x x	0	1 1

因为指令数少，可以直接不简化的直接使用表达式表示各个控制信号（也可以用case穷举）

控制单元实现

```
`timescale 1ns / 1ps

module cu(
    input [5:0]funct,op,
    input eq,

    //第一类：选择器信号
    output regdst,alusrc,shift,m2reg,jal,
    output [1:0]pcsrc,
```

```
    //第二类：存储器寄存器写使能
    output wmem,wreg,
    //第三类：alu
    output [3:0]aluc,
    //第四类
    output sext
);
    wire rtype=~op[5]&~op[4]&~op[3]&~op[2]&~op[1]&~op[0];//op 000000
    wire i_add=rtype&funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 100000
    wire i_sub=rtype&funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];//op 0 funct 100010
    wire i_and=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&~funct[0];//op 0 funct 100100
    wire i_or=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct[1]&funct[0];//op 0 funct 100101
    wire i_xor=rtype&funct[5]&~funct[4]&~funct[3]&funct[2]&funct[1]&~funct[0];//op 0 funct 100110
    wire i_sll=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 000000
    wire i_srl=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0];//op 0  funct 000010
    wire i_sra=rtype&~funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&funct[0];//op 0 funct 000011
    wire i_jr=rtype&~funct[5]&~funct[4]&funct[3]&~funct[2]&~funct[1]&~funct[0];//op 0 funct 001000
    wire i_addi=~op[5]&~op[4]&op[3]&~op[2]&~op[1]&~op[0];//op 001000
    wire i_andi=~op[5]&~op[4]&op[3]&op[2]&~op[1]&~op[0];//op 001100
    wire i_ori=~op[5]&~op[4]&op[3]&op[2]&~op[1]&op[0];//op 001101
    wire i_xori=~op[5]&~op[4]&op[3]&op[2]&op[1]&~op[0];//op 001110
    wire i_lw=op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];//op 100011
    wire i_sw=op[5]&~op[4]&op[3]&~op[2]&op[1]&op[0];////op 101011
    wire i_beq=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&~op[0];//op 000100
    wire i_bne=~op[5]&~op[4]&~op[3]&op[2]&~op[1]&op[0];//op 000101
    wire i_lui=~op[5]&~op[4]&op[3]&op[2]&op[1]&op[0];//op 001111
    wire i_j=~op[5]&~op[4]&~op[3]&~op[2]&op[1]&~op[0];//op 000010
    wire i_jal=~op[5]&~op[4]&~op[3]&~op[2]&op[1]&op[0];//op 000011

    assign
wreg=i_add|i_sub|i_add|i_or|i_xor|i_sll|i_srl|i_sra|i_addi|i_andi|i_ori|i_xori|i_lw|i_lui|i_jal;
    assign regdst=i_addi|i_andi|i_ori|i_xori|i_lw|i_lui;
```

```
    assign jal=i_jal;
    assign m2reg=i_lw;
    assign shift=i_sll|i_srl|i_sra;
    assign alusrc=i_addi|i_andi|i_ori|i_xori|i_lw|i_sw+i_lui;
    assign sext=i_addi|i_lw|i_sw|i_beq|i_bne;
    assign aluc[3]=i_sra;
    assign aluc[2]=i_sub|i_or|i_srl|i_sra|i_ori|i_lui;
    assign aluc[1]=i_xor|i_sll|i_srl|i_sra|i_xori|i_beq|i_bne|i_lui;
    assign aluc[0]=i_and|i_or|i_sll|i_srl|i_sra|i_andi|i_ori;
    assign wmem=i_sw;
    assign pcsrc[1]=i_j|i_jal|i_jr;
    assign pcsrc[0]=i_beq&eq|i_bne&~eq|i_j|i_jal;
endmodule
```

Verilog

5.5 存储器实现+仿真

这里不采用IP核实现存储器，而是采用设计代码去实现

指令存储器inst_rom

```
`timescale 1ns / 1ps

module inst_rom #(parameter LENGTH = 32)(
    input [31:0]pc,
    output [31:0]inst
);
    wire [31:0]rom[LENGTH-1:0];
    assign rom[5'h00] = 32'h3c010000; // (00) main: lui $1, 0
    assign rom[5'h01] = 32'h34240050; // (04) ori $4, $1, 80
    assign rom[5'h02] = 32'h20050004; // (08) addi $5, $0, 4
    assign rom[5'h03] = 32'h0c000018; // (0c) call: jal sum
    assign rom[5'h04] = 32'hac820000; // (10) sw $2, 0($4)
    assign rom[5'h05] = 32'h8c890000; // (14) lw $9, 0($4)
```

```
    assign rom[5'h06] = 32'h01244022; // (18) sub $8, $9, $4
    assign rom[5'h07] = 32'h20050003; // (1c) addi $5, $0, 3
    assign rom[5'h08] = 32'h20a5ffff; // (20) loop2: addi $5, $5, -1
    assign rom[5'h09] = 32'h34a8ffff; // (24) ori $8, $5, 0xffff
    assign rom[5'h0A] = 32'h39085555; // (28) xori $8, $8, 0x5555
    assign rom[5'h0B] = 32'h2009ffff; // (2c) addi $9, $0, -1
    assign rom[5'h0C] = 32'h312affff; // (30) andi $10,$9, 0xffff
    assign rom[5'h0D] = 32'h01493025; // (34) or $6, $10, $9
    assign rom[5'h0E] = 32'h01494026; // (38) xor $8, $10, $9
    assign rom[5'h0F] = 32'h01463824; // (3c) and $7, $10, $6
    assign rom[5'h10] = 32'h10a00001; // (40) beq $5, $0, shift
    assign rom[5'h11] = 32'h08000008; // (44) j loop2
    assign rom[5'h12] = 32'h2005ffff; // (48) shift: addi $5, $0, -1
    assign rom[5'h13] = 32'h000543c0; // (4c) sll $8, $5, 15
    assign rom[5'h14] = 32'h00084400; // (50) sll $8, $8, 16
    assign rom[5'h15] = 32'h00084403; // (54) sra $8, $8, 16
    assign rom[5'h16] = 32'h000843c2; // (58) srl $8, $8, 15
    assign rom[5'h17] = 32'h08000017; // (5c) finish: j finish
    assign rom[5'h18] = 32'h00004020; // (60) sum: add $8, $0, $0
    assign rom[5'h19] = 32'h8c890000; // (64) loop: lw $9, 0($4)
    assign rom[5'h1A] = 32'h20840004; // (68) addi $4, $4, 4
    assign rom[5'h1B] = 32'h01094020; // (6c) add $8, $8, $9
    assign rom[5'h1C] = 32'h20a5ffff; // (70) addi $5, $5, -1
    assign rom[5'h1D] = 32'h14a0fffb; // (74) bne $5, $0, loop
    assign rom[5'h1E] = 32'h00081000; // (78) sll $2, $8, 0
    assign rom[5'h1F] = 32'h03e00008; // (7c) jr $31
    assign inst = rom[pc[6:2]]; // use word address to read rom,pc每次+4
endmodule
```

Verilog

数据存储器data_ram

```Verilog
`timescale 1ns / 1ps

module data_ram #(parameter LENGTH = 32)(
    input clk,we,

    input [31:0]addr,wdata,
    output [31:0]readData
);
    reg [31:0]ram[LENGTH-1:0];
    assign readData = ram[addr[6:2]]; // use word address to read ram
    always @(posedge clk) begin
        if (we) begin
            ram[addr[6:2]]<=wdata;
        end
    end

    //RAM初始化
    integer i;
    initial begin // initialize memory
        for (i = 0; i < 32; i = i + 1)
            ram[i] = 0;
        // ram[word_addr] = data // (byte_addr) item in data array
        ram[5'h14] = 32'h000000a3; // (50) data[0] 0 + A3 = A3 80字节处
        ram[5'h15] = 32'h00000027; // (54) data[1] a3 + 27 = ca
        ram[5'h16] = 32'h00000079; // (58) data[2] ca + 79 = 143
        ram[5'h17] = 32'h00000115; // (5c) data[3] 143 + 115 = 258
        // ram[5'h18] should be 0x00000258, the sum stored by sw instruction
    end
endmodule
```

Verilog

集成

```
`timescale 1ns / 1ps

module top(
    input clk,clrn,
    output [31:0]pc,
    output [31:0]inst,
    output [31:0]aluout,memread
);
    wire [5:0]funct,op;
    wire eq;
    wire regdst,alusrc,shift,m2reg,jal;
    wire [1:0]pcsrc;
    wire wmem,wreg;
    wire [3:0]aluc;
    wire sext;
    wire [31:0]qb;
    datapath  datapath_inst (
        .clk(clk),
        .clrn(clrn),
        .regdst(regdst),
        .alusrc(alusrc),
        .shift(shift),
        .m2reg(m2reg),
        .jal(jal),
        .wreg(wreg),
        .sext(sext),
        .pcsrc(pcsrc),
        .aluc(aluc),
        .memread(memread),
        .inst(inst),
        .eq(eq),
```

```
        .funct(funct),
        .op(op),
        .pc(pc),
        .aluout(aluout),
        .qb(qb)
    );
    cu  cu_inst (
        .funct(funct),
        .op(op),
        .eq(eq),
        .regdst(regdst),
        .alusrc(alusrc),
        .shift(shift),
        .m2reg(m2reg),
        .jal(jal),
        .pcsrc(pcsrc),
        .wmem(wmem),
        .wreg(wreg),
        .aluc(aluc),
        .sext(sext)
    );
    inst_rom inst_romInst(pc,inst);
    data_ram data_ramInst(clk,wmem,aluout,qb,memread);
endmodule
```

Verilog

仿真

```
`timescale 1ns / 1ps

module top_test();
    reg clk,clrn;
    wire [31:0]pc;
```

```
    wire [31:0]inst,aluout,memread;

    initial begin
        clk=0;clrn=0;
        #3;clrn=1;
        #4;clrn=0;
    end

    always #5 clk=~clk;
    top top_init(clk,clrn,pc,inst,aluout,memread);
endmodule
```

Verilog

> 注意：

1. datapath内对funct、op的赋值
2. jal写31号寄存器的数据选择，寄存器选择
3. 端口命名要一致，处理也要一致

[注释1] ALU、寄存器文件、多路选择器等

[注释2] 指令数少