7 多周期无流水线CPU设计 MC CPU

英语

1. *regardless*不管、不考虑

7.1将指令的执行分为几个时钟周期

MC CPU设计的关键点是**将指令的执行分为了“IF、ID、EXE、MEM、WB”这五个阶段，每一个阶段都经过一个时钟周期**[注释1]，**每个时钟周期产生赋值该阶段用到的控制信号**[注释2]。但是**并不是所有的指令都需要执行完全部的这五个阶段**，因此我们就可以使用更少的时钟周期去执行简单的指令，用较多的时钟周期执行复杂的指令[注释3]

最复杂的指令是lw指令，需要经过5个阶段，历时5个时钟周期

逻辑、算术指令和sw指令只需要“IF、ID、EXE、WB”这4个阶段，历时4个时钟周期

分支指令是在ID段计算分支目标地址bpc、在EXE阶段判断是否需要跳转，经过“IF、ID、MEM”三个阶段，历时3个时钟周期

无条件跳转指令，只经过“IF、ID”这两个阶段，历时2个时钟周期

下图为SC CPU和MC CPU执行“lw、beq、add、j”指令序列，所需要的时间

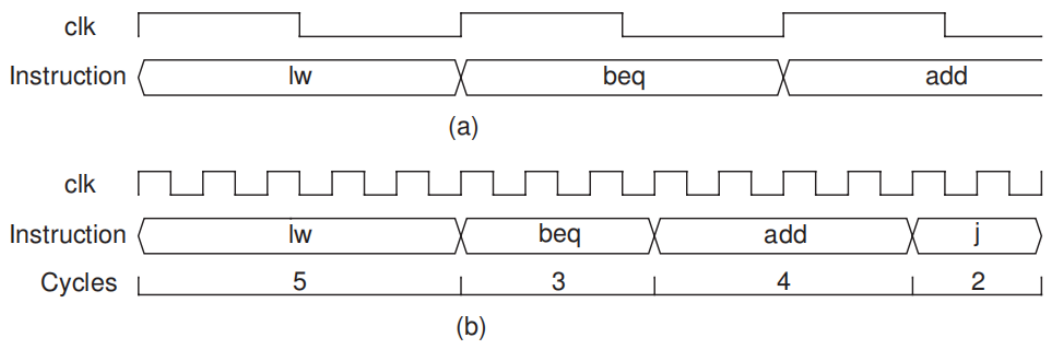

Figure 7.1 Timing comparison of (a) single-cycle and (b) multiple-cycle CPUs

📌**由此可见，MC CPU较于SC CPU虽然复杂度增加，但是时间消耗更少**

7.1.1取指阶段

取指阶段所做的工作如下：

```
IR <-- Memory[PC];
PC <-- PC + 4;
```

因此需要IR、PC两个寄存器，它们的写使能信号分别是wir、wpc

这里的PC+4我们选择使用ALU来实现PC+4，其示意图如下：

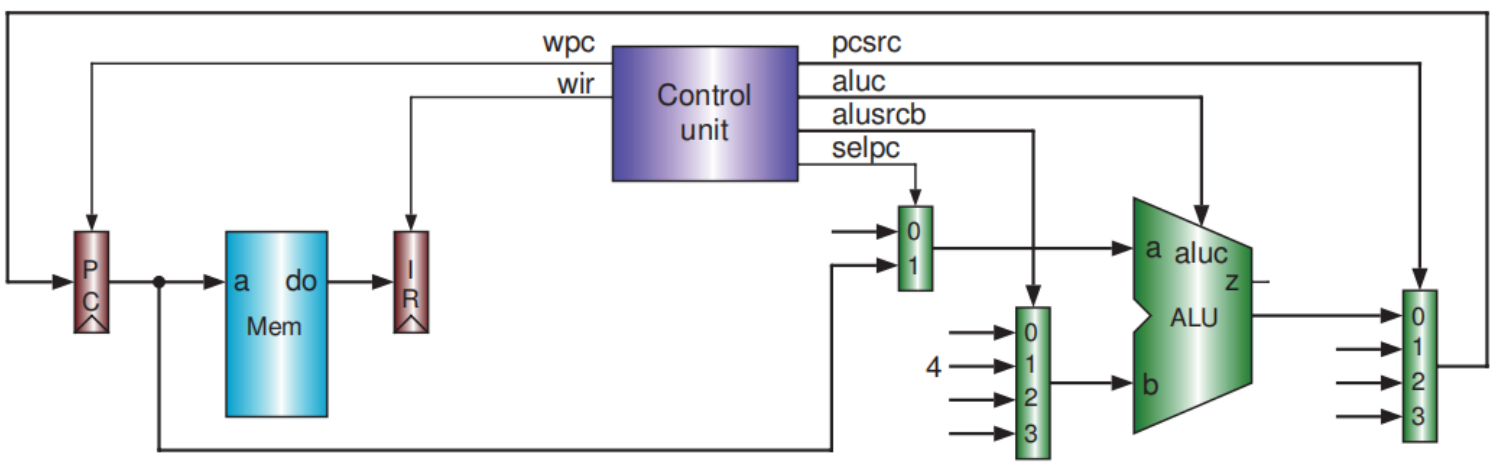

Figure 7.2 Block diagram of instruction fetch of an MC CPU

这一阶段用到的控制信号有：PC写使能wpc、IR写使能wir、ALU的a端口选择selpc、ALU的b端口选择alusrcb和ALU的运算控制信号aluc以及NPC选择pcsrc

7.1.2译码和读操作数阶段

非跳转指令的ID阶段

非跳转指令是所有的逻辑、算术指令和分支指令，这些指令都**需要读操作数（读rs、rt、立即数等），进行立即数的扩展；分支指令则需要计算偏移地址**

```
A <-- RegisterFile[rs];
B <-- RegisterFile[rt];
C <-- PC + sign_extend(offset) << 2;
```

I型指令则需要扩展立即数

需要注意的是**I型逻辑运算指令将立即数零扩展，其他I型指令都是有符号数扩展**，用sext信号选择实现

下图为非跳转指令的译码阶段示意图：

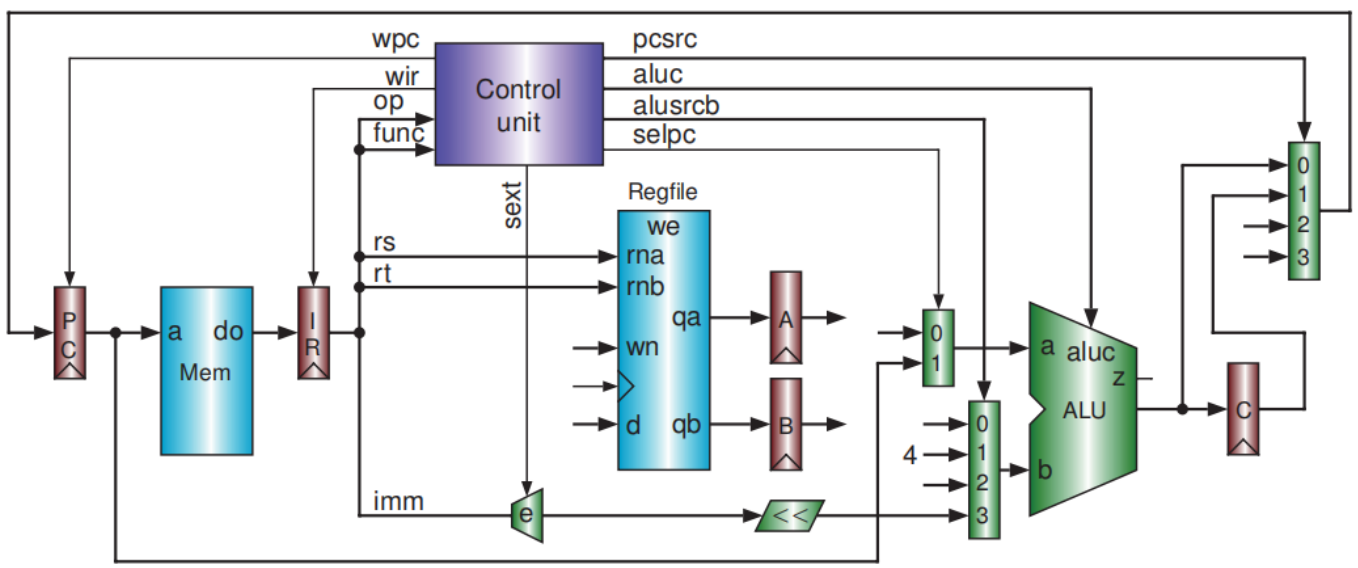

Figure 7.3 Block diagram of instruction decode of an MC CPU

这一状态用到的控制信号有：sext、alusrcb、selpc、aluc

跳转指令的ID阶段

跳转指令的ID阶段则是获取到跳转地址，然后写PC

Jal、J指令的跳转地址是{pcAdder[31:28],指令[26:0],2'b0}，Jal还需要将PC+4写进$31寄存器

Jr指令的跳转地址是由指令中的rs寄存器所指定的

```
j:    PC <-- {PC[31:28],address,00};
jal:  RegisterFile[31] <-- PC;
      PC <-- {PC[31:28],address,00};
jr:   PC <-- RegisterFile[rs];
```

下图为跳转指令在ID阶段所做的工作

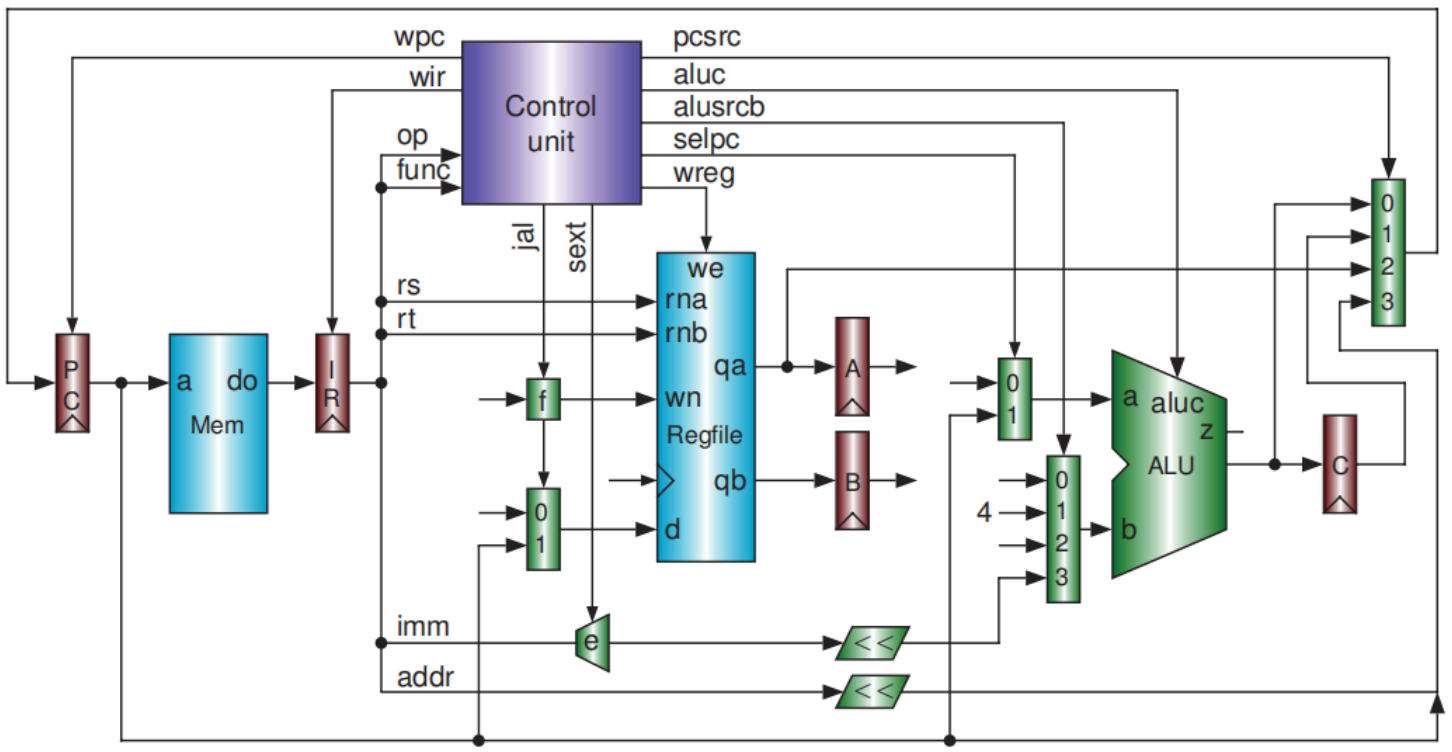

Figure 7.4 Block diagram for jump instructions in an MC CPU

用到的控制信号有：wpc、wir、jal、sext、wreg、pcsrc

7.1.3执行阶段

分支指令

在执行阶段，beq、bne这些分支指令需要**进行分支条件的判断，然后根据判断结果更新PC**

beq、bne指令的判断条件是"[rs]是否等于[rt]"，**若相等则beq可以实现跳转，若不等则bne可以实现跳转**

```
beq: If (A == B) PC <-- C;
bne: If (A != B) PC <-- C;
```

寄存器C中存储的是ID阶段计算出的目标地址

下图为分支指令执行阶段的示意图，这一阶段用到的控制信号有：wpc、wir、aluc、selpc、alusrcb、pcsrc

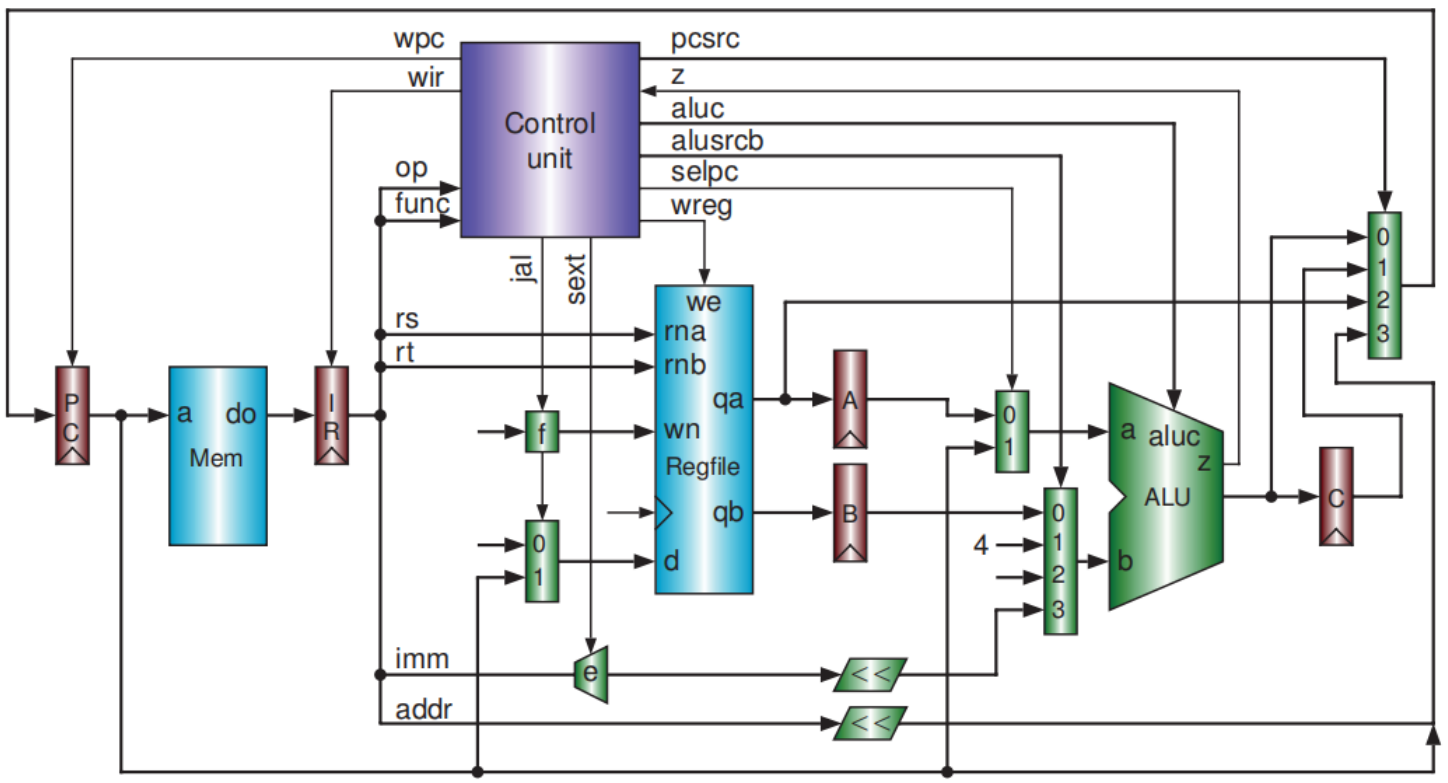

Figure 7.5 Block diagram for branch instructions in an MC CPU

非分支指令

对于非分支指令，**CPU根据aluc，对ALU a、b端口的操作数做相应的操作，并将结果写入寄存器C**。下图为这些非分支指令所完成的操作的Behavioral表达式

```
add/sub/and/or/xor: C <-- A op B;
sll:                C <-- B << sa;
srl:                C <-- B >> sa;
sra:                C <-- signed(B) >>> sa;
addi:               C <-- A +  sign_extend(immediate);
andi/ori/xori:      C <-- A op zero_extend(immediate);
lw/sw:              C <-- A +  sign_extend(offset);
lui:                C <-- immediate << 16;
```

对于add、sub、and、or、xor这五条指令，它们只有aluc不同，shift=0,selpc=0,alusrcb=00

对于sll、srl、sra这三条移位指令，除aluc不同外,shift=1,selpc=0,alusrcb=00

addi、lw、sw这三条I型指令，均是rs+signed(Imm),aluc=x000,shift=0,pcsel=0,alucsrcb=10

andi、ori、xori这三条I型逻辑运算指令，是将rs与无符号数扩展后的立即数运算shift=0,selpc=0,alusrcb=10

lui指令是将立即数左移16位写入寄存器C，不需要去在意扩展的方式，alusrcb=10

lw、sw指令的功能实现同addi

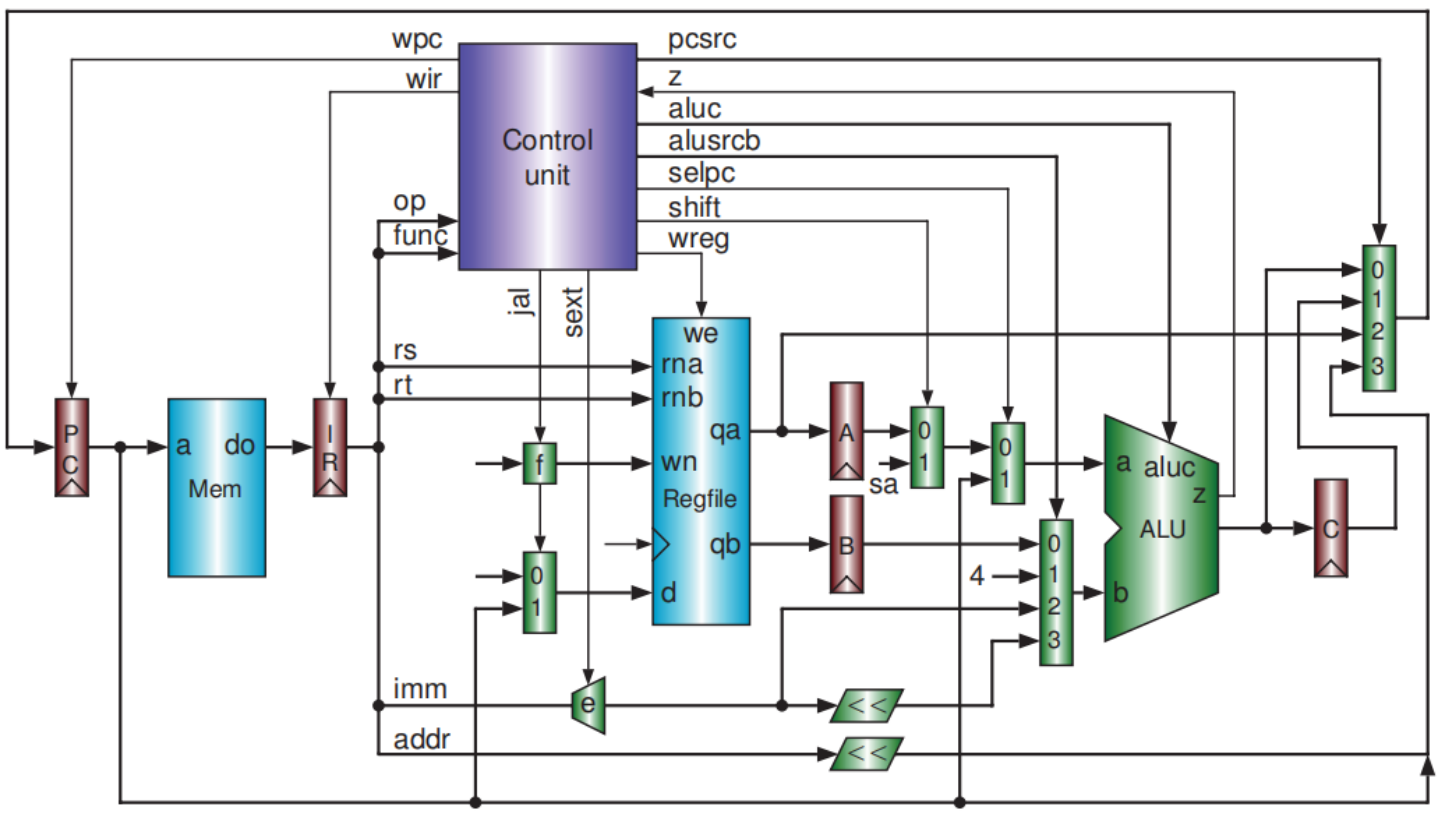

Figure 7.6 Block diagram for computational instructions in an MC CPU

7.1.4访存阶段

访存阶段只有lw、sw指令工作，两指令均使用运算结果C中内容作为访存地址，前者从dataMem中读取数据，后者将qb写入dataMem

```
lw: DR <-- Memory[C];
sw: Memory[C] <-- B;
```

下图为lw、sw指令在mem阶段的示意图，用到的控制信号有wmem、iord(分离存储就不需要这个信号)

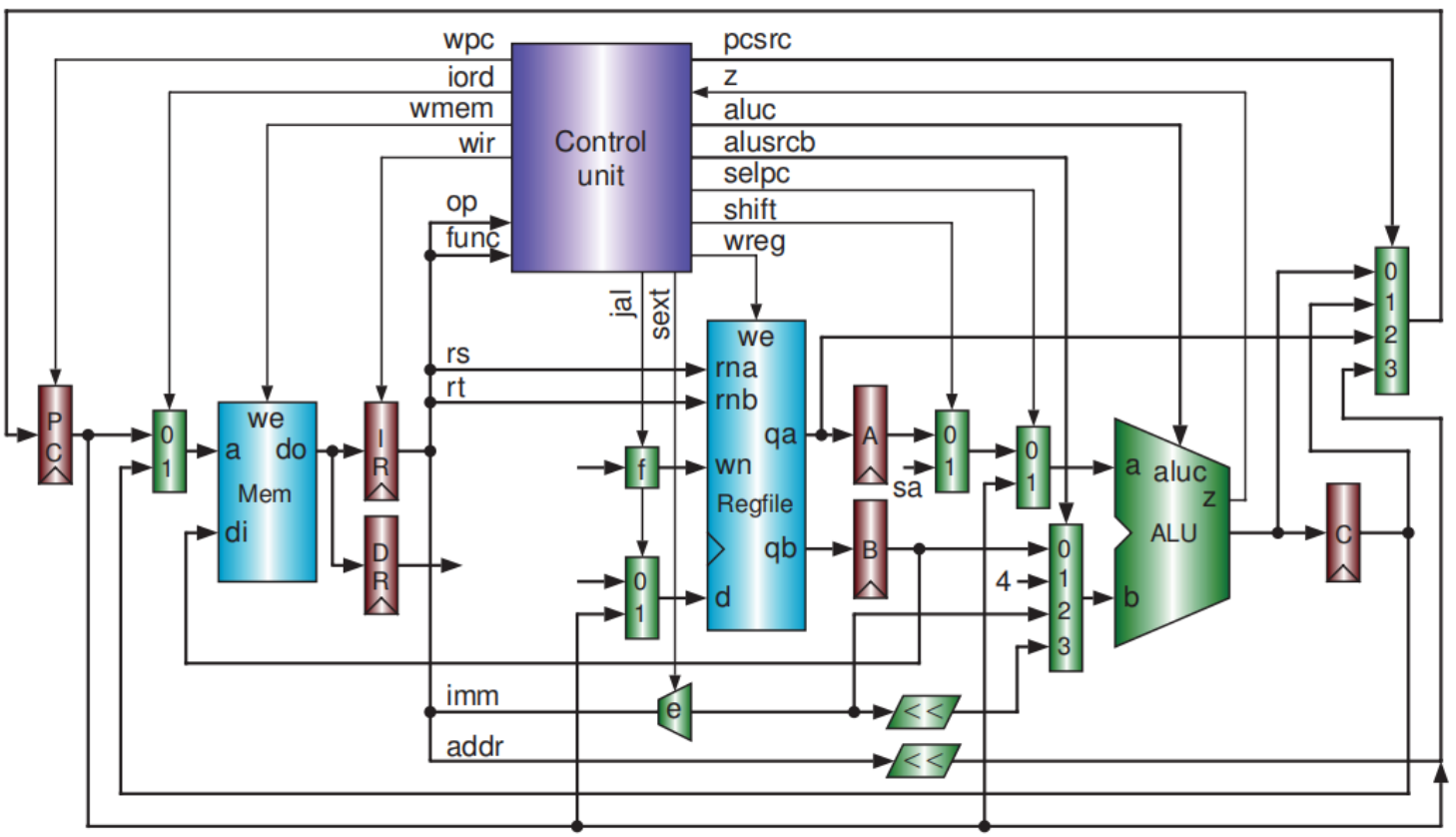

Figure 7.7 Block diagram for load and store instructions in an MC CPU

7.1.5写回阶段

写回阶段即根据选择信号m2reg选择写lw读存储器结果DR还是运算指令的运算结果C到目标寄存器；目标寄存器也有两种：R型指令写rd，I型指令写rt，根据regdst信号做选择

```
add/sub/and/or/xor/sll/srl/sra: RegisterFile[rd] <-- C;
addi/andi/ori/xori/lui:         RegisterFile[rt] <-- C;
lw:                             RegisterFile[rt] <-- DR;
```

下图为写回阶段的电路示意图

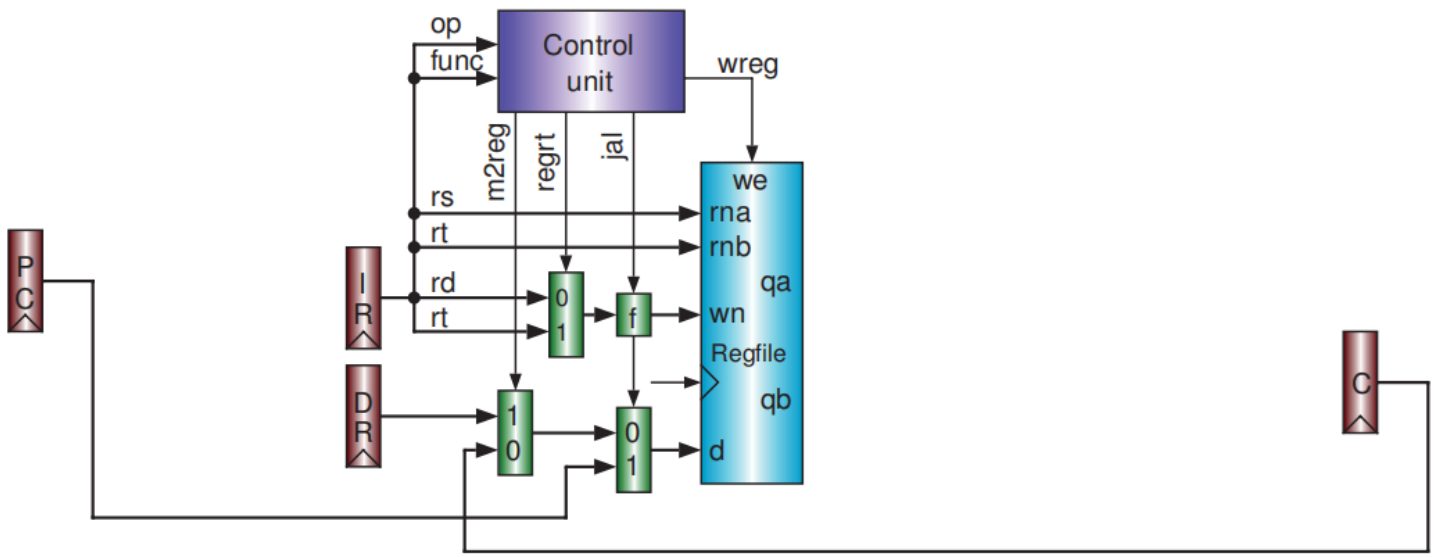

Figure 7.8 Block diagram of write back of an MC CPU

7.2 MC CPU示意图和数据通路实现

7.2.1 MC CPU结构

MC CPU的整体结构如下图所示，采用指令和数据共同存储的方式，只使用一个存储器，mccpu则集成了控制器模块和数据通路模块

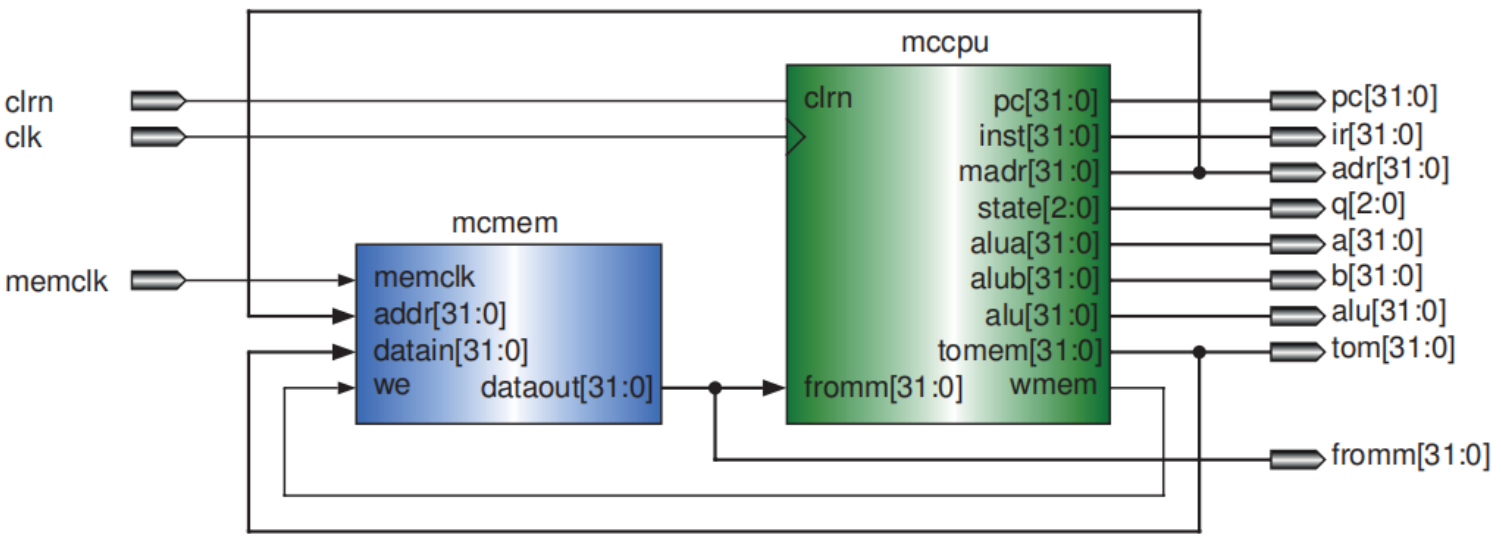

Figure 7.10 Block diagram of a MC computer

7.2.2 数据通路模块的实现

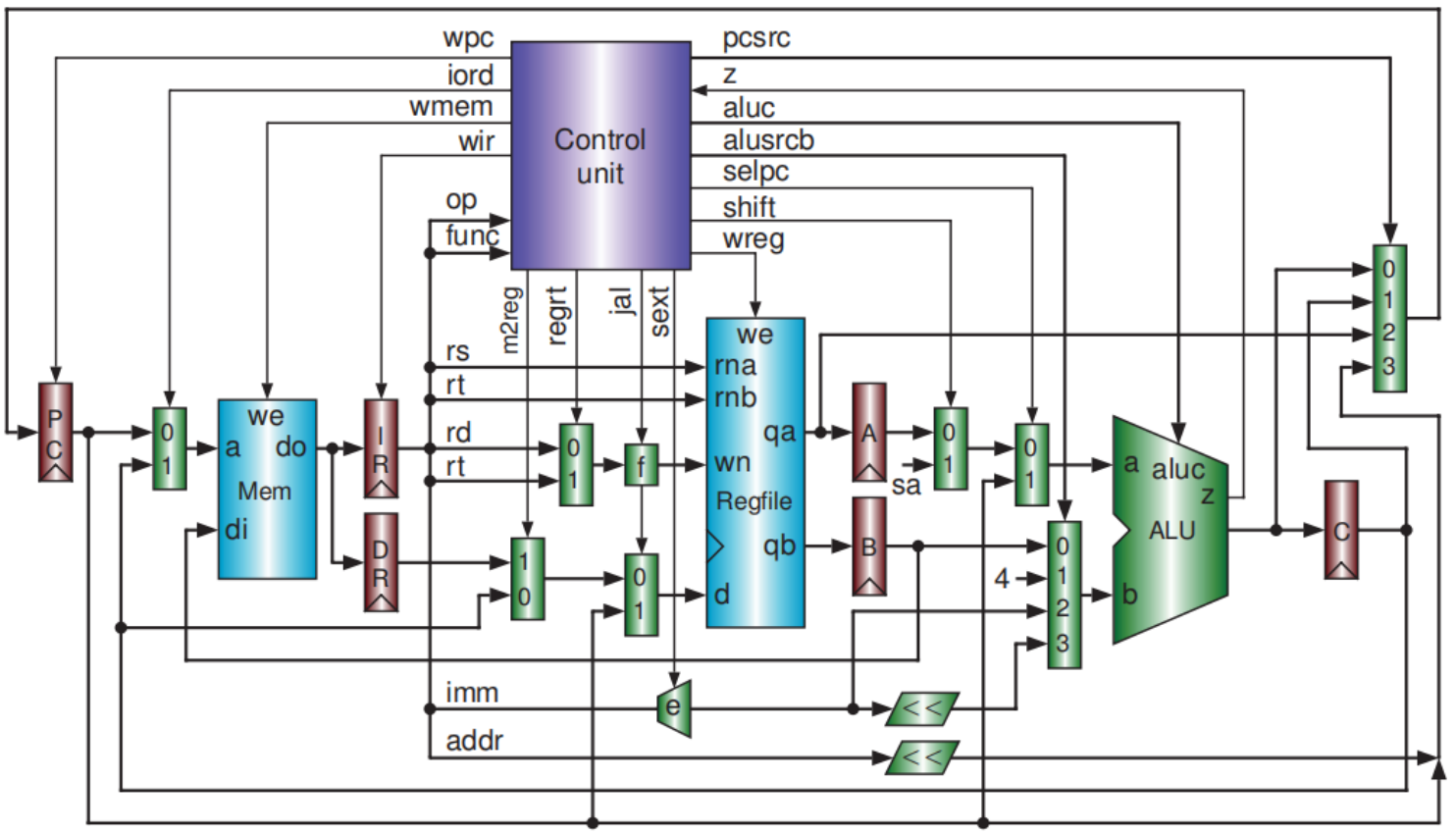

Figure 7.9 Block diagram of an MC CPU plus memory

```
`timescale 1ns / 1ps

module datapath(
    input clk,clrn,
    input regdst,alusrc,shift,m2reg,jal,iord,selpc,
    input wreg,wpc,wir,
    input sext,

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

    input [31:0]memread,

    output eq,
    output [5:0]funct,op,
```

```
    output [31:0]qb,memaddr
);
    //取指阶段
    //pc赋值
    wire [31:0]npc,aluout,c,jpc,rpc,pc;
    wire [31:0]inst,data;
    mux4to1 npc_init(aluout,c,rpc,jpc,pcsrc,npc);
    pc pc_init(clk,clrn,wpc,npc,pc);
    assign jpc={pc[31:29],inst[25:0],2'b00};
    ir ir_init(clk,clrn,wir,memread,inst);
    reg_design dr_init(clk,1'b1,memread,data);

    //译码阶段
    //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(rd,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(c,data,m2reg,wregData_temp);
    wire [31:0]wregData;
```

```
    mux2to1 generate_wregData(wregData_temp,pc,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
    wire [15:0]imm=inst[15:0];
    wire [31:0]imm_sext={{16{sext&imm[15]}},imm};
    wire [31:0]imm_sext_left={imm_sext[29:0],2'b0};

    //ALU模块
    wire [31:0]a_temp,a,b;
    wire z,overflow;
    mux2to1 generate_aluaTemp(qa,{26'b0,imm[10:6]},shift,a_temp);
    mux2to1 generate_alua(a_temp,pc,selpc,a);
    mux4to1 generate_alub(qb,32'd4,imm_sext,imm_sext_left,alusrcb,b);
    alu alu_init(a,b,aluc,aluout,z,overflow);

    reg_design C(clk,1'b1,aluout,c);
    mux2to1 generate_memaddr(pc,c,iord,memaddr);

endmodule
```

Verilog

7.3 MC CPU控制器模块的实现

📌 **在MC CPU中，不同的指令需要不同的时钟周期去执行，这可以通过有穷状态机实现**

MC CPU的CU结构如下：

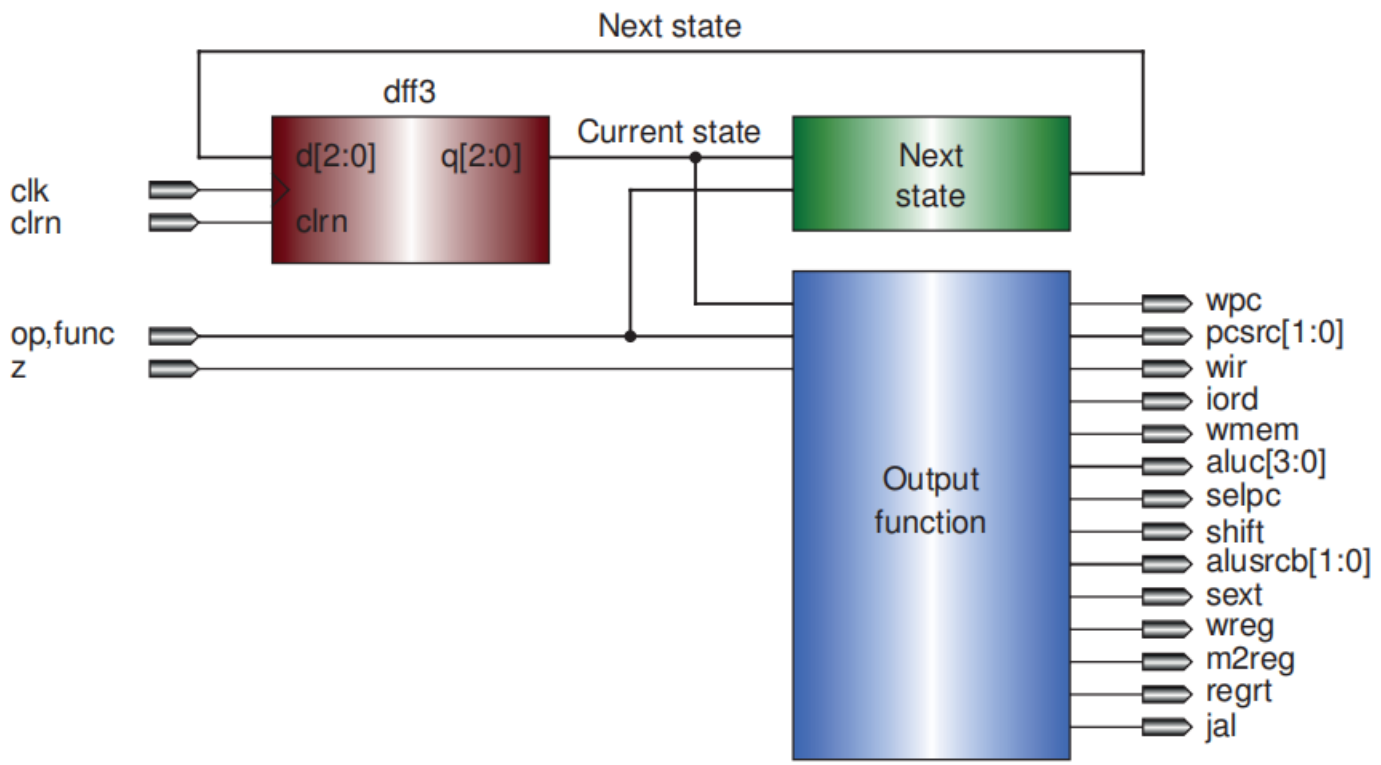

Figure 7.12 Block diagram of the control unit of an MC CPU

其中状态的传递用时序逻辑实现，下一个状态的确定以及输出都用组合逻辑实现[注释4]

7.3.1 控制单元的状态转移

我们将指令的执行划分为五个阶段，那么就有5个状态，分别用“000、001、010、011、100”表示IF、ID、EX、MEM和WB，共需要3位的寄存器位去传递

状态转移分析如下；

所有的指令都会从000→001

只有j、jr、jal指令会从001→000，其余指令都会从001→010

只有beq、bne指令会从010→000，只有lw、sw指令会从010→011，其余指令都从010→100

只有sw指令会从011→000，lw指令会从011→100

进入100的指令都会从100→000

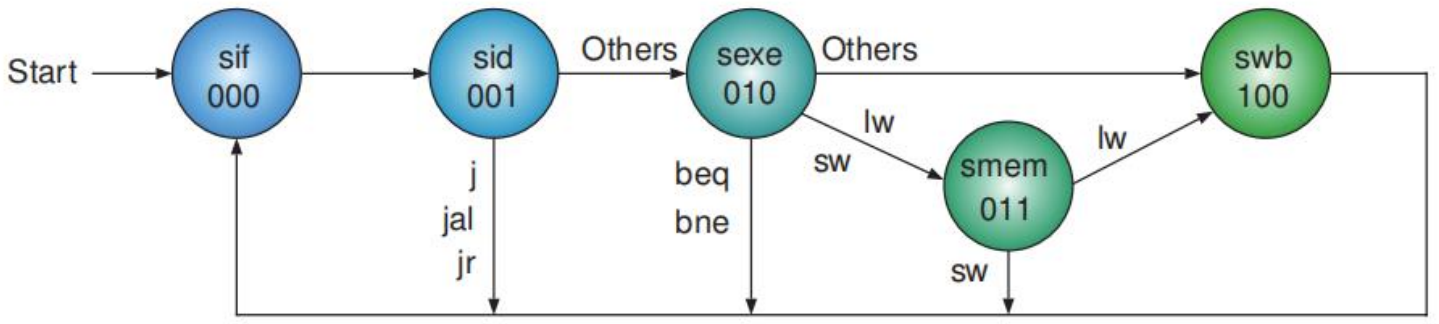

Figure 7.11 State transition diagram of an MC CPU

7.3.2下一个状态的确定

Table 7.2 Truth table of next state function

Current state		Input	Next state	
State	q[2:0]	Instruction	State	d[2:0]
sif	000	x	sid	001
sid	001	i_j	sif	000
		i_jal	sif	000
		i_jr	sif	000
		Others	sexe	010
sexe	010	i_beq	sif	000
		i_bne	sif	000
		i_lw	smem	011
		i_sw	smem	011
		Others	swb	100
smem	011	i_lw	swb	100
		i_sw	sif	000
swb	100	x	sif	000

根据7.3.1得到的状态转移关系，我们可以得到左侧的关于下一个状态的真值表

从这个表中，也可以方便的利用Behav风格设计组合逻辑电路确定下一个状态

7.3.3输出的确定

CU的输出即为控制信号，根据7.1对各个阶段的分析，我们可以得到下面的每个阶段对控制信号再赋值的真值表

Input			Output													

State	Inst	z	wpc	pcsrc[1:0]	wir	iord	wmem	aluc[3:0]	selpc	shift	alusrcb[1:0]	sext	wreg	m2reg	regdst	jal	初始时所有信号为0
if	x	x	1	0	1	0	0	0	1	0	1	0	0	0	0	0	取指/PC+4->PC
id	i_j	x	1	11	0	0	0	0	0	0	0	0	0	0	0	0	跳转
	i_jal	x	1	11	0	0	0	0	0	0	0	0	1	0	0	1	跳转并写寄存器
	i_jr	x	1	10	0	0	0	0	0	0	0	0	0	0	0	0	按寄存器跳转
	others	x	0	0	0	0	0	x000	1	0	11	1	0	0	0	0	计算分支地址
exe	i_add	x	0	0	0	0	0	x000	0	0	0	0	0	0	0	0	
	i_sub	x	0	0	0	0	0	x100	0	0	0	0	0	0	0	0	
	i_and	x	0	0	0	0	0	x001	0	0	0	0	0	0	0	0	
	i_or	x	0	0	0	0	0	x101	0	0	0	0	0	0	0	0	
	i_xor	x	0	0	0	0	0	x010	0	0	0	0	0	0	0	0	
	i_sll	x	0	0	0	0	0	11	0	1	0	0	0	0	0	0	
	i_srl	x	0	0	0	0	0	111	0	1	0	0	0	0	0	0	
	i_sra	x	0	0	0	0	0	1111	0	1	0	0	0	0	0	0	
	i_addi	x	0	0	0	0	0	x000	0	0	10	1	0	0	0	0	
	i_andi	x	0	0	0	0	0	x100	0	0	10	0	0	0	0	0	
	i_ori	x	0	0	0	0	0	x101	0	0	10	0	0	0	0	0	
	i_xori	x	0	0	0	0	0	x010	0	0	10	0	0	0	0	0	
	i_lw	x	0	0	0	0	0	x000	0	0	10	1	0	0	0	0	
	i_sw	x	0	0	0	0	0	x000	0	0	10	1	0	0	0	0	
	i_beq	1	1	1	0	0	0	x010	0	0	0	0	0	0	0	0	
	i_beq	0	0	0	0	0	0	x010	0	0	0	0	0	0	0	0	
	i_bne	1	0	0	0	0	0	x010	0	0	0	0	0	0	0	0	
	i_bne	0	1	1	0	0	0	x010	0	0	0	0	0	0	0	0	

	i_lui	x	0	0	0	0	0	x110	0	0	10	0	0	0	0	0	
mem	i_lw	x	0	0	0	1	0	0	0	0	0	0	0	0	0	0	
	i_sw	x	0	0	0	1	1	0	0	0	0	0	0	0	0	0	
wb	rtype	x	0	0	0	0	0	0	0	0	0	0	1	0	0	0	
	i_andi	x	0	0	0	0	0	0	0	0	0	0	1	0	1	0	
	i_ori	x	0	0	0	0	0	0	0	0	0	0	1	0	1	0	
	i_lui	x	0	0	0	0	0	0	0	0	0	0	1	0	1	0	
	i_lw	x	0	0	0	0	0	0	0	0	0	0	1	1	1	0	
	i_xori	x	0	0	0	0	0	0	0	0	0	0	1	0	1	0	
	i_addi	x	0	0	0	0	0	0	0	0	0	0	1	0	1	0	

7.3.4 Verilog代码实现

```
`timescale 1ns / 1ps

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

    //第一类：选择器信号
    output reg regdst,alusrc,shift,m2reg,jal,iord,selpc,
    output reg [1:0]pcsrc,alusrcb,
    //第二类：存储器寄存器写使能
    output reg wmem,wreg,wpc,wir,
    //第三类：alu
    output reg [3:0]aluc,
    //第四类
    output reg 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

    reg [2:0]cur_state,next_state;

    //时序逻辑传递状态
    always @(posedge clk or posedge clrn) begin
        if (clrn) begin
            cur_state<=3'b000;
        end else begin
            cur_state<=next_state;
        end
    end
```

```
    //组合逻辑确定状态
    always @(*) begin
        case(cur_state)
            3'b000:next_state=3'b001;
            3'b001:begin
                if (i_j|i_jal|i_jr) begin
                    next_state=3'b000;
                end else begin
                    next_state=3'b010;
                end
            end
            3'b010:begin
                if (i_beq|i_bne) begin
                    next_state=3'b000;
                end else if (i_lw|i_sw) begin
                    next_state=3'b011;
                end else begin
                    next_state=3'b100;
                end
            end
            3'b011:begin
                if (i_sw) begin
                    next_state=3'b000;
                end else begin
                    next_state=3'b100;
                end
            end
            3'b100:next_state=3'b000;
        endcase
    end
    //组合逻辑确定输出
    always @(*) begin
```

```
        wpc=0;pcsrc=2'b0;wir=0;iord=0;wmem=0;aluc=4'b0;
        selpc=0;shift=0;alusrcb=2'b0;sext=0;wreg=0;m2reg=0;regdst=0;jal=0;
        case (cur_state)
            3'b000:begin
                wpc=1;wir=1;
                selpc=1;alusrcb=2'b1;
            end
            3'b001:begin
                if (i_j) begin
                    wpc=1;pcsrc=2'b11;
                end else if (i_jal) begin
                    wpc=1;pcsrc=2'b11;wreg=1;jal=1;
                end else if (i_jr) begin
                    wpc=1;pcsrc=2'b10;
                end else begin
                    aluc=4'b0000;selpc=1;alusrcb=2'b11;sext=1;
                end
            end
            3'b010:begin
                aluc[3] = i_sra;
                aluc[2] = i_sub | i_or | i_srl | i_sra | i_ori | i_lui;
                aluc[1] = i_xor | i_sll | i_srl | i_sra | i_xori | i_beq | i_bne | i_lui;
                aluc[0] = i_and | i_or | i_sll | i_srl | i_sra | i_andi | i_ori;
                if (i_sll|i_srl|i_sra) begin
                    shift=1;
                end
                if (i_addi|i_andi|i_ori|i_xori|i_lw|i_sw|i_lui) begin
                    alusrcb=2'b10;
                end
                if (i_addi|i_lw|i_sw) begin
                    sext=1;
                end
                if (i_beq&eq|i_bne&~eq) begin
```

```
                wpc=1;pcsrc=2'b1;
            end
        end
        3'b011:begin
            if (i_lw) begin
                iord=1;
            end else if (i_sw) begin
                wmem=1;
                iord=1;
            end
        end
        3'b100:begin
            wreg=1;
            if (~rtype) begin
                regdst=1;
            end
            if (i_lw) begin
                m2reg=1;
            end
        end
    endcase
  end
endmodule
```

Verilog

7.4存储器设计

MC CPU采用“指令和数据聚合存储”，只用到了一个存储器

```
`timescale 1ns / 1ps

module mem(
```

```
    input clk,
    input[31:0]addr,idata,
    input we,
    output [31:0]odata
);
    reg [31:0]memory[63:0];
    initial begin
        memory[0] = 32'h3c010000;
        memory[1] = 32'h34240080;
        memory[2] = 32'h20050004;
        memory[3] = 32'h0c000018;
        memory[4] = 32'hac820000;
        memory[5] = 32'h8c890000;
        memory[6] = 32'h01244022;
        memory[7] = 32'h20050003;
        memory[8] = 32'h20a5ffff;
        memory[9] = 32'h34a8ffff;
        memory[10] = 32'h39085555;
        memory[11] = 32'h2009ffff;
        memory[12] = 32'h312affff;
        memory[13] = 32'h01493025;
        memory[14] = 32'h01494026;
        memory[15] = 32'h01463824;
        memory[16] = 32'h10a00001;
        memory[17] = 32'h08000008;
        memory[18] = 32'h2005ffff;
        memory[19] = 32'h000543c0;
        memory[20] = 32'h00084400;
        memory[21] = 32'h00084403;
        memory[22] = 32'h000843c2;
        memory[23] = 32'h08000017;
        memory[24] = 32'h00004020;
        memory[25] = 32'h8c890000;
```

```
        memory[26] = 32'h20840004;
        memory[27] = 32'h01094020;
        memory[28] = 32'h20a5ffff;
        memory[29] = 32'h14a0fffb;
        memory[30] = 32'h00081000;
        memory[31] = 32'h03e00008;
        memory[32] = 32'h000000a3;
        memory[33] = 32'h00000027;
        memory[34] = 32'h00000079;
        memory[35] = 32'h00000115;
        memory[36] = 32'h00000000;
    end
    assign odata=memory[addr[7:2]];
    always @(posedge clk) begin
        if (we) begin
            memory[addr[7:2]]<=idata;
        end
    end
endmodule
```

Verilog

7.5顶层模块+测试仿真

7.5.1顶层模块

```
`timescale 1ns / 1ps

module top(
    input clk,clrn,
    output [31:0]memread,
    output [31:0]memaddr
);
    wire regdst,alusrc,shift,m2reg,jal,iord,selpc;
```

```
    wire wreg,wpc,wir;
    wire sext;

    wire [1:0]pcsrc,alusrcb;
    wire [3:0]aluc;

    wire eq;
    wire [5:0]funct,op;
    wire [31:0]qb;
    datapath  datapath_inst (
        .clk(clk),
        .clrn(clrn),
        .regdst(regdst),
        .alusrc(alusrc),
        .shift(shift),
        .m2reg(m2reg),
        .jal(jal),
        .iord(iord),
        .selpc(selpc),
        .wreg(wreg),
        .wpc(wpc),
        .wir(wir),
        .sext(sext),
        .pcsrc(pcsrc),
        .alusrcb(alusrcb),
        .aluc(aluc),
        .memread(memread),
        .eq(eq),
        .funct(funct),
        .op(op),
        .qb(qb),
        .memaddr(memaddr)
    );
```

```
    cu  cu_inst (
        .clk(clk),
        .clrn(clrn),
        .funct(funct),
        .op(op),
        .eq(eq),
        .regdst(regdst),
        .alusrc(alusrc),
        .shift(shift),
        .m2reg(m2reg),
        .jal(jal),
        .iord(iord),
        .selpc(selpc),
        .pcsrc(pcsrc),
        .alusrcb(alusrcb),
        .wmem(wmem),
        .wreg(wreg),
        .wpc(wpc),
        .wir(wir),
        .aluc(aluc),
        .sext(sext)
    );
    mem  mem_inst (
        .clk(clk),
        .addr(memaddr),
        .idata(qb),
        .we(wmem),
        .odata(memread)
    );
endmodule
```

Verilog

7.5.2测试

```
`timescale 1ns / 1ps

module top_test();
    reg clk,clrn;
    wire [31:0]memread;
    wire [31:0]memaddr;

    initial begin
        clk=0;clrn=0;
        #3;clrn=1;
        #1;clrn=0;
    end
    always #5 clk=~clk;
    top  top_inst (
        .clk(clk),
        .clrn(clrn),
        .memread(memread),
        .memaddr(memaddr)
    );
endmodule
```

Verilog

习题

1. *Compare SC CPUs and MC CPUs. Indicate their advantages and disadvantages.*

   SC CPU最经济有效，最简单，但是最耗时

   MC CPU较SC CPU复杂，但是时间花费相对减少

2. *Can the registers A, B, and DR in Figure 7.9 be deleted?*

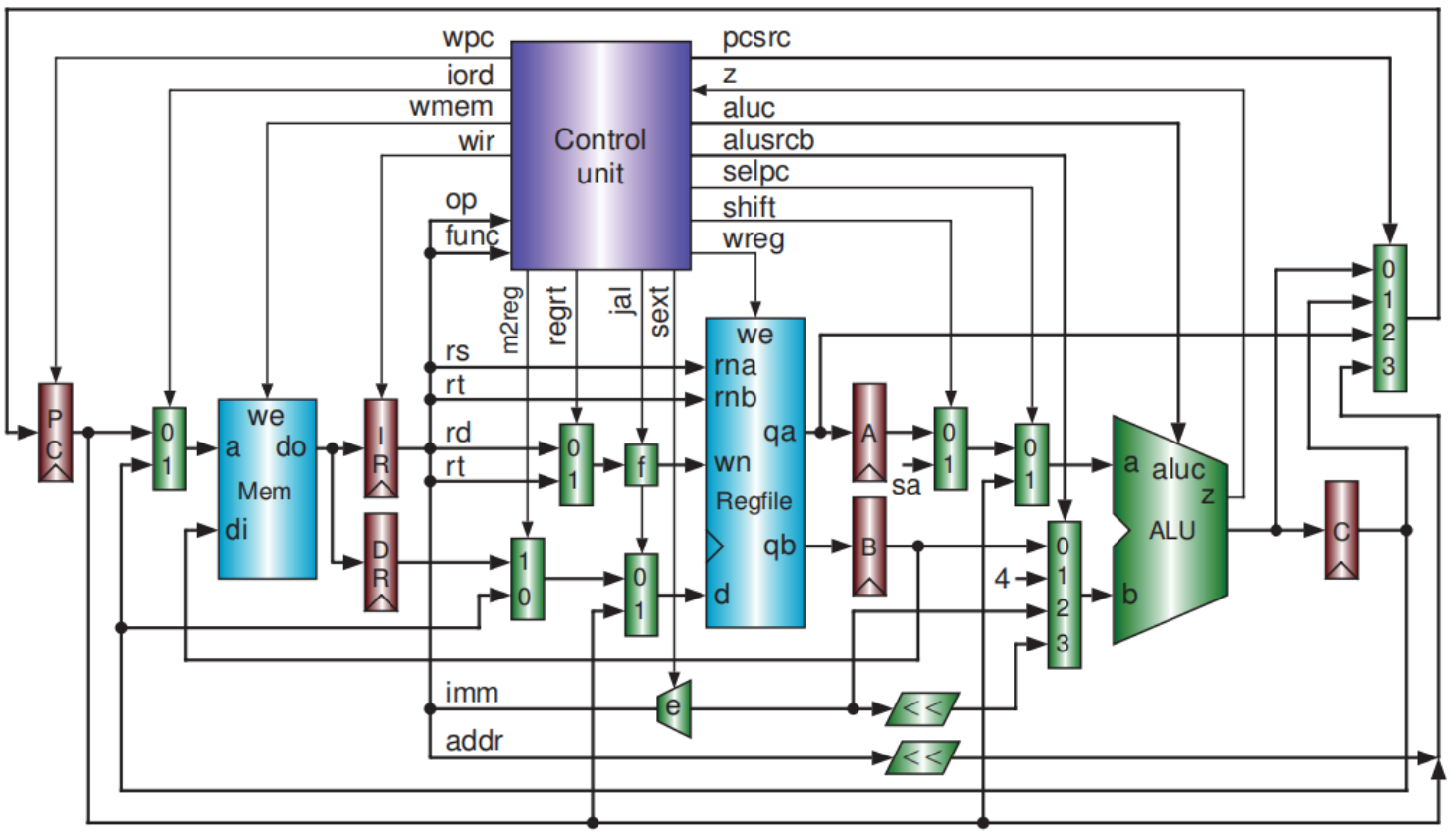

Figure 7.9 Block diagram of an MC CPU plus memory

是可以删除A、B、DR的，因为IR在执行期间不变，只有在IF时才会写IR，那么指令就不会改变，rs、rt就不会改变，从而并不需要A、B去寄存数据；而DR是需要的，因为在MEM时已经得到了存储器数据，但是此时m2reg并不有效（到WB时才有效），因此需要保存数据到WB阶段

3. *Design an MC CPU in Verilog HDL of the behavioral style. Requirement: the lui instruction takes only two clock cycles.*

LUI只需要两个时钟周期即可完成，那么它的状态图就需要改变，**和JAL的处理一样**，**写寄存器的数据可以在ID阶段产生，此阶段此指令生成wreg、regdst=1，再增加一个控制信号lui，若此控制信号有效，则选择写移位结果**

4. *Design an MC CPU with the state transition diagram of Figure 7.16.*

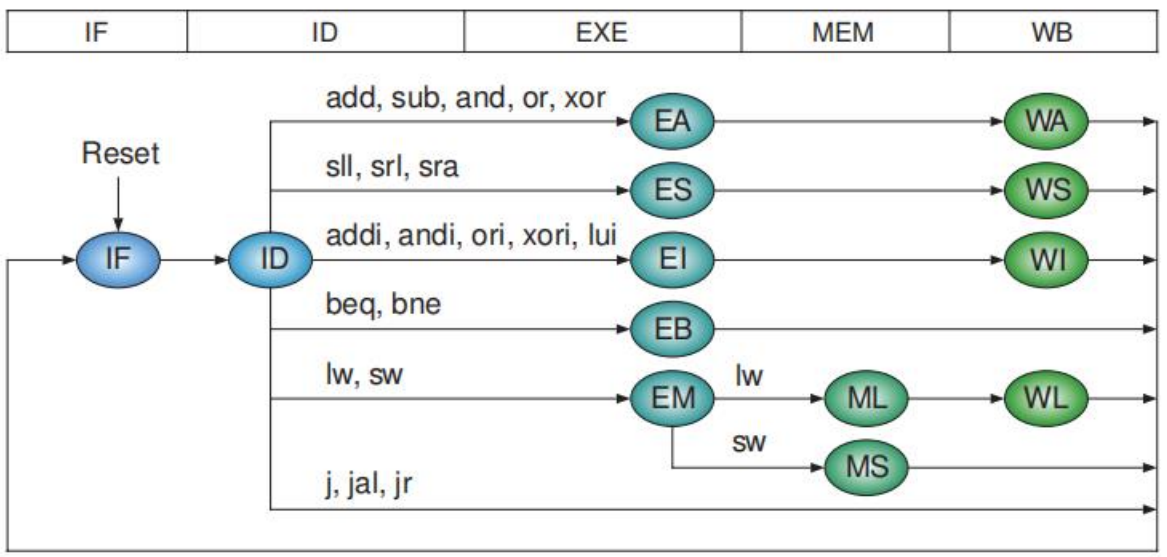

Figure 7.16 Another state transition diagram of an MC CPU

状态的更新不变、下一状态的产生需要改变，输出也需要改变，主要改变CU模块

5. **实现具有中断/异常控制的MC CPU**

[注释1] 这个时钟周期可以一样，便于流水；也可以不一样

[注释2] 这样控制信号就可以看作CU有穷状态机的输出

[注释3] MC CPU的好处

[注释4] 三段式的话是输出也用时序逻辑