第 3 章　单周期 CPU 的设计与实现

3.1　实 验 介 绍

本次实验使用 Verilog HDL 实现 54 条 MIPS 指令的 CPU 的设计和仿真，本次设计的 CPU 是单周期的，即每条指令均在一个周期内完成。首先要明确本次 MIPS CPU 的设计是建立在指令集（Instruction Set Architecture，ISA）的基础上的，ISA 告诉 CPU 设计者 CPU 应该做什么，而告诉编译器开发者 CPU 可以做什么。

本次实验要从 54 条指令开始，熟悉每条指令的功能和格式，确定所需要的部件模块，画出相应的数据通路（可能是某一类指令的数据通路），最后合成总的数据通路，分析确定出所有控制信号并对每条指令进行验证。

实验目标：

- 深入了解 CPU 的原理。
- 熟悉并掌握 54 条指令的格式和数据通路。
- 画出实现 54 条指令的 CPU 的通路图。
- 学习使用 Verilog HDL 设计实现 54 条指令的 CPU。

3.2　总 体 设 计

3.2.1　指令格式

（1）理解需要实现的 54 条 MIPS 指令，每条指令的介绍如表 3.1 所示。

表 3.1　54 条指令格式表

指令	指 令 说 明	指 令 格 式	OP 31-26	FUNCT 5-0	指令码十六进制
addi	加立即数	addi rt,rs,immediate	001000		20000000
addiu	加立即数（无符号）	addiu rd,rs,immediate	001001		24000000
andi	立即数与	andi rt,rs,immediate	001100		30000000
ori	或立即数	ori rt,rs,immediate	001101		34000000
sltiu	小于立即数置 1（无符号）	sltiu rt,rs,immediate	001011		2C000000
lui	立即数加载高位	lui rt,immediate	001111		3C000000
xori	异或（立即数）	xori rt,rs,immediate	001110		38000000
slti	小于置 1（立即数）	slti rt,rs,immediate	001010		28000000
addu	加（无符号）	addu rd,rs,rt	000000	100001	00000021
and	与	and rd,rs,rt	000000	100100	00000024
beq	相等时分支	beq rs,rt,offset	000100		10000000

续表

指令	指令说明	指令格式	OP 31-26	FUNCT 5-0	指令码十六进制
bne	不等时分支	bne rs,rt,offset	000101		14000000
j	跳转	j target	000010		08000000
jal	跳转并链接	jal target	000011		0C000000
jr	跳转至寄存器所指地址	jr rs	000000	001000	00000009
lw	取字	lw rt,offset(base)	100011		8C000000
xor	异或	xor rd,rs,rt	000000	100110	00000026
nor	或非	nor rd,rs,rt	000000	100111	00000027
or	或	or rd,rs,rt	000000	100101	00000025
sll	逻辑左移	sll rd,rt,sa	000000	000000	00000000
sllv	逻辑左移(位数可变)	sllv rd,rt,rs	000000	000100	00000004
sltu	小于置1(无符号)	sltu rd,rs,rt	000000	101011	0000002B
sra	算数右移	sra rd,rt,sa	000000	000011	00000003
sri	逻辑右移	sri rd,rt,sa	000000	000010	00000002
subu	减(无符号)	subu rd,rs,rt	000000	100010	00000022
sw	存字	sw rt,offset(base)	101011		AC000000
add	加	add rd,rs,rt	000000	100000	00000020
sub	减	sub rd,rs,rt	000000	100010	00000022
slt	小于置1	slt rd,rs,rt	000000	101010	0000002A
srlv	逻辑右移(位数可变)	srlv rd,rt,rs	000000	000110	00000006
srav	算数右移(位数可变)	srav rd,rt,rs	000000	000111	00000007
clz	前导零计数	clz rd,rs	011100	100000	70000020
divu	除(无符号)	divu rs,rt	000000	011011	0000001B
eret	异常返回	eret	010000	011000	42000018
jalr	跳转至寄存器所指地址,返回地址保存在引号寄存器	jalr rs	000000	001001	00000008
lb	取字节	lb rt,offset(base)	100000		80000000
lbu	取字节(无符号)	lbu rt,offset(base)	100100		90000000
lhu	取半字(无符号)	lhu rt,offset(base)	100101		94000000
sb	存字节	sb rt,offset(base)	101000		A0000000
sh	存半字	sh rt,offset(base)	101001		A4000000
lh	取半字	lh rt,offset(base)	100001		84000000
mfc0	读CPO寄存器	mfc0 rt,rd	010000	000000	40000000
mfhi	读Hi寄存器	mfhi rd	000000	010000	00000010
mflo	读Lo寄存器	mflo rd	000000	010010	00000012
mtc0	写CPO寄存器	mtc0 rt,rd	010000	000000	40800000
mthi	写Hi寄存器	mthi rd	000000	010001	00000011

续表

指令	指令说明	指令格式	OP 31-26	FUNCT 5-0	指令码十六进制
mtlo	写 Lo 寄存器	mtlo rd	000000	010011	00000013
mul	乘	mul rd,rs,rt	011100	000010	70000002
multu	乘(无符号)	multu rs,rt	000000	011001	00000019
syscall	系统调用	syscall	000000	001100	0000000C
teq	相等异常	teq rs,rt	000000	110100	00000034
bgez	大于或等于 0 时分支	bgez rs,offset	000001		04010000
break	断点	break	000000	001101	0000000D
div	除	div rs,rt	000000	011010	0000001A

(2) 进行以下步骤。

① 阅读每条指令,充分了解每条指令所需执行的功能与过程 。以 addi 指令为例,该指令将寄存器 rs 里的值与立即数 imm 相加,结果放到寄存器 rt 里,注意是符号数的相加。

```
addi rt,rs,imm  #reg[rt]=reg[rs]+(sign)imm
```

② 画出每条指令在执行过程中的数据通路图。pc 寄存器的值作为地址传到指令寄存器,同时 pc+4 放回 pc 寄存器;指令寄存器取出指令,将指令译码,op 和 func 字段送到控制单元,rs 送入通用寄存器组的 ra 接口,取出相应寄存器的值由 qa 输出,而立即数进行符号位的位拓展;ALU 对输入的两个数进行加运算,结果送回通用寄存器组。addi 指令数据通路图如图 3.1 所示。可以观察到,此数据通路图适用于很多的 R 类型指令,只改变某些输入端口和控制信号即可,如指令 addiu、xori、ori 等。

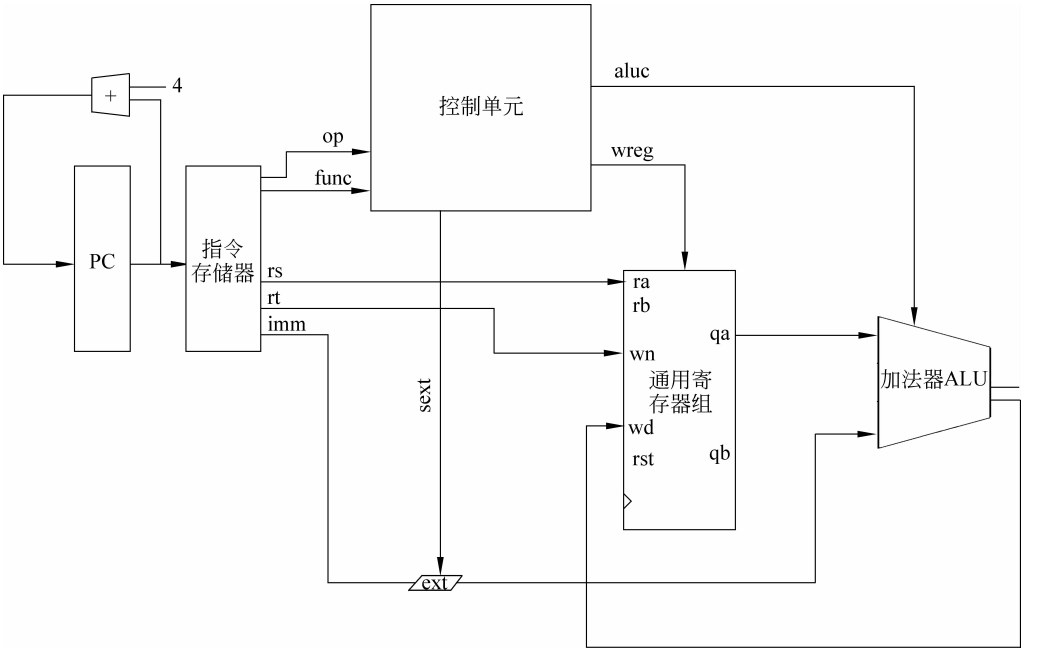


图 3.1 addi 指令数据通路图

③ 考虑所有指令，增加多路选择器，画出整个数据通路。因为不同的指令和操作会造成很多模块有不同的输入，此时需要多路选择器判断哪个数据应当进入输入端，控制信号由控制单元产生，如图 3.2 所示，通用寄存器组写地址端口 wn，它与输入的指令类型有关，当指令是 R 类型时，wn＝rd；当指令是 i 类型时，wn＝rt；当指令是 jal 时，wn＝5'b11111（返回地址放在 31 号通用寄存器）。将第二步所有指令的数据通路图联通到一起，得到一个整体的数据通路图，如图 3.3 简化版所示。

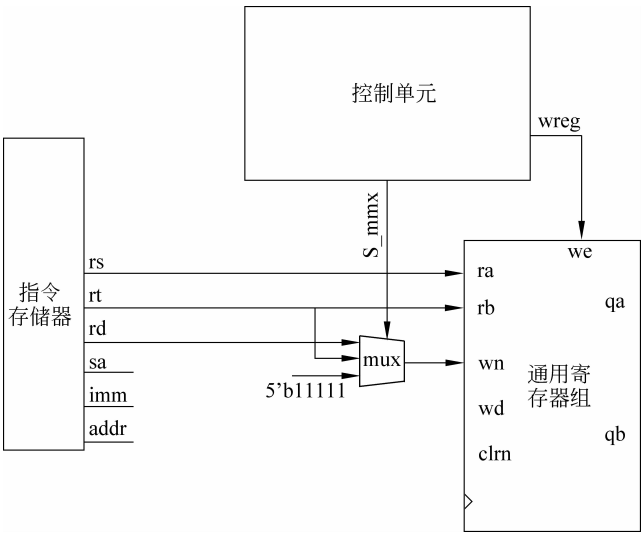


图 3.2　一个 mux 例子

④ 列出真值表，确定控制信号，完成控制模块。由第三步画出的数据通路图，得到所有的控制信号，列出真值表。例如，表 3.2 为五条指令的真值表。

表 3.2　五条指令的真值表

	wreg	reg_rt	jal	shift	sext	aluc[3:0]	pcsrc[1:0]	…
addi	1	1	0	0	1	x000	00	
and	1	0	0	0	0	x001	00	
sll	1	0	0	1	0	0011	00	
jal	1	0	1	x	x	xxxx	11	
jr	0	x	0	x	x	xxxx	10	
…								

⑤ 编写各个模块代码，进行指令验证。

3.2.2　总体数据通路

画出每一条或每一类指令的数据通路图，最后组合成总体的设计图。图 3.3 为简化版总体数据通路设计图。

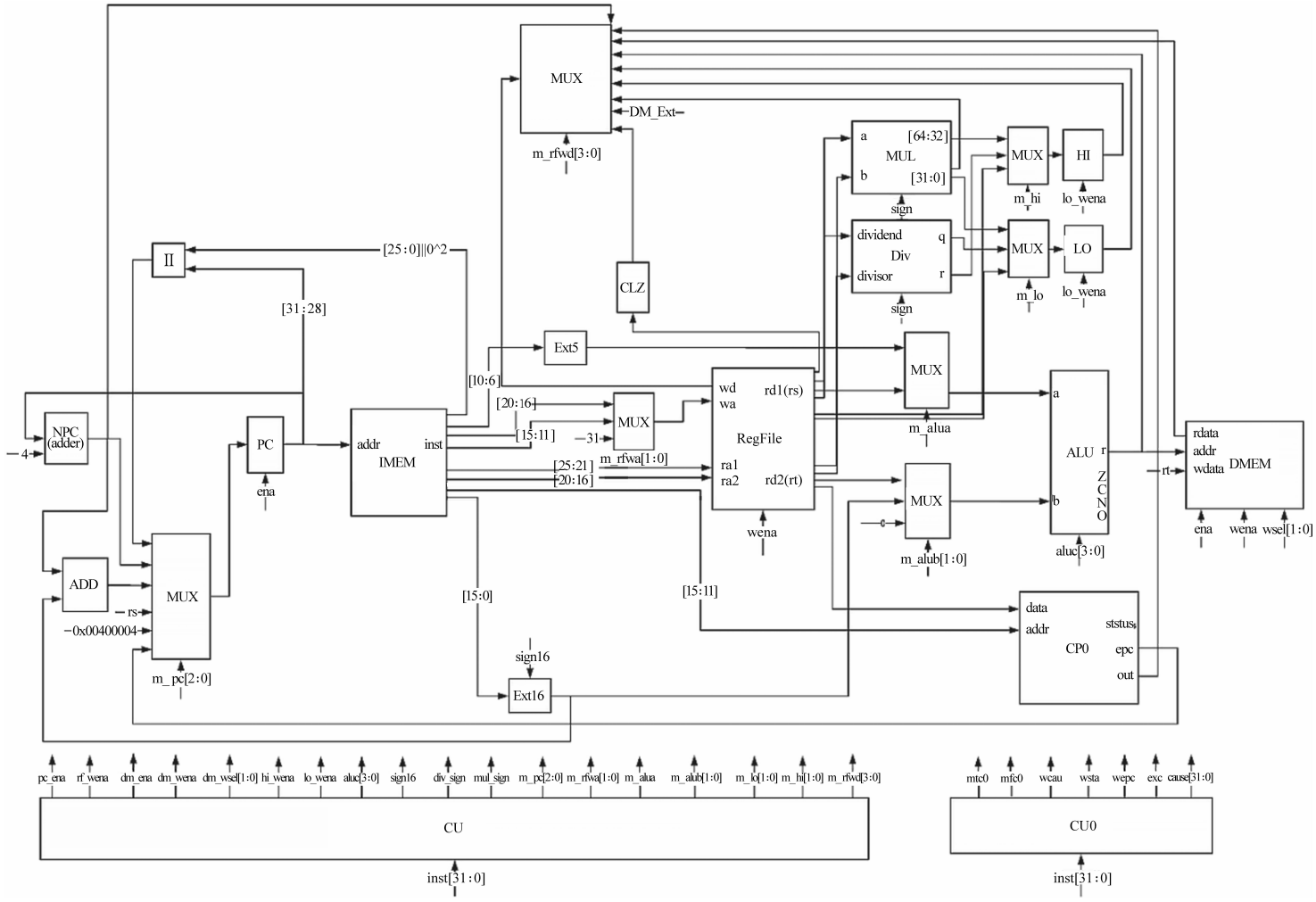


图 3.3　简化版总体数据通路设计图

3.3 主要模块设计

下面给出本次实验设计所需的主要模块,也可以根据自己的设计习惯完成 CPU 的组织架构。

(1) 顶层模块 sccomp_dataflow.v,将各个模块拼接起来。

```
module sccomp_dataflow(
    input clk_in,
    input rst,
    input [15:0] sw,
    output [7:0] o_seg,
    output [7:0] o_sel
);
wire [31:0]pc,inst,addr;
wire reset=~rst;
wire [31:0]dm_rdata;          //从数据存储器读出的数据
wire [31:0]dm_wdata;          //写入数据存储器的数据
wire [3:0]dm_wena;            //数据存储器写信号
wire [31:0]pc_in;             //待写入 pc 寄存器的值
wire [31:0]addr_in;           //待写入指令寄存器的值
wire [4:0]cp0_addr;           //读或写 cp0 的地址
wire [31:0]cp0_wdata,status,epc,cp0_rdata;
wire [31:0]rdata,wdata;       // 数据寄存器的读和写数据
assign pc_in=pc-32'h00400000;
assign addr_in=addr -32'h10010000;

//分频器,采用 IP 核 clk_ip
clk_ip cpu_clk(clk_in,clk,reset);

//中央处理器
cpu sccpu(clk,reset,inst,rdata,epc,cp0_rdata,pc,dm_ena,dm_wena,
addr,wdata,cp0_addr,cp0_wdata);

//指令存储器,采用 IP 核 imem_ip
imem_ip imem(ip_in[12:2],inst);

//数据存储器
dmem scdmem(clk,dm_ena,dm_wena[0],dm_wena,addr-32'h10010000,wdata,dm_rdata);
cp cp(clk,reset,pc,inst,cp0_addr,cp0_wdata,status,epc,cp0_rdata);  //cp 协处理器
io_sel io_mem(addr, dm_wena, dm_ena, seg7_cs, switch_cs);  //地址映射部件选择部件
seg7x16 seg7(clk, reset, seg7_cs, wdata, o_seg, o_sel);     //数码管显示部件
sw_mem_sel sw_mem(switch_cs, sw, dm_rdata, rdata);          //sw 指令写入选择部件
endmodule
```

(2) cpu 模块 cpu.v。

```
module cpu(
    input clk,
    input reset,
    input [31:0] inst,                          //imem 读出的指令
    input [31:0] dm_rdata,                      //dmem 读出的数据
    input [31:0] cp0_epc,                       //产生异常的 pc 位置
    input [31:0] cp0_rdata,
    output [31:0] pc,
    output dm_ena,                              //存储器有效信号
    output dm_wena,                             //存储器写有效信号
    output [31:0] dm_addr,                      //存储器的读写地址
    output [31:0] dm_wdata,                     //写入存储器的数据
    output [1:0] dm_wsel,
    output [4:0] cp0_addr,                      //cp0 的读写地址
    output [31:0] cp0_wdata                     //写入 cp0 的数据
);
wire [3:0]aluc;
wire [31:0]alua,alub,aluz;
wire [31:0]pc_in,npc_out;wire [4:0]rf_waddr;
wire [31:0]rf_wdata,rf_rdata1,rf_rdata2;

wire [31:0]ext16_32;
wire [31:0]hi_in,hi_out,lo_in,lo_out;
wire [31:0]clz_out,div_q,div_r,mul_hi,mul_lo;
wire [2:0]m_pc;
wire [1:0]m_alub,m_hi,m_lo,m_rfwa;
wire [3:0]m_rfwd;
assign cp0_addr=inst[15:11];
assign cp0_wdata=rf_rdata2;
assign dm_addr=aluz;
assign dm_wdata=rf_rdata2;

pcreg pcreg(clk,reset,pc_ena,pc_in,pc);       //pc 寄存器模块
regfile cpu_ref(clk,reset,rf_wena,rf_waddr,rf_wdata,inst[25:21];    //通用寄存器模块
inst[20:16],rf_rdata1,rf_rdata2);adder npc(pc,4,npc_out);
alu alu(alua,alub,aluc,aluz,zero,carry,negative,overflow);       //alu 模块
ext16 ext16(inst[15:0],sign16,ext16_32);      //立即数扩展模块
clz clz(rf_rdata1,clz_out);                   //运算器模块,用于计算前导 0 的数目

div div(rf_rdata1,rf_rdata2,div_sign,div_q,div_r);          //除法器
mul mul(rf_rdata1,rf_rdata2,mul_sign,mul_hi,mul_lo);        //乘法器
regi hi(clk,reset,hi_wena,hi_in,hi_out);                    //hi 寄存器
regi lo(clk,reset,lo_wena,lo_in,lo_out);
```

```
//32 位 8 选 1 多路选择器,选择下一个 pc 的值
mux8 mux_pc(m_pc,{pc[31:28],inst[25:0],2'b00},npc_out,npc_out+(ext16_32<<2),
    rf_rdata1,32'h00400004,cp0_epc,0,0,pc_in);

//32 位 4 选 1 多路选择器,选择要写入通用寄存器的地址
mux4 mux_rfwa(m_rfwa,{27'b0,inst[20:16]}, {27'b0,inst[15:11]},31,0,rf_waddr);
mux mux_alua(m_alua,rf_rdata1,{27'b0,inst[10:6]},alua);
mux4 mux_alub(m_alub,rf_rdata2,ext16_32,0,0,alub);
mux4 mux_hi(m_hi,rf_rdata1,mul_hi,div_r,0,hi_in);
mux4 mux_lo(m_lo,rf_rdata1,mul_lo,div_q,0,lo_in);

//32 位 16 选 1 多路选择器,选择要写入通用寄存器的数据
mux16 mux_rfwd(m_rfwd,npc_out,dm_rdata,aluz,mul_lo,{24'b0,dm_rdata[7:0]},
    {16'b0,dm_rdata[15:0]},{{24{dm_rdata[7]}},dm_rdata[7:0]},{{16{dm_rdata
    [15]}},dm_rdata[15:0]},hi_out,lo_out,cp0_rdata,clz_out,0,0,0,0,rf_wdata);

//控制器,输出各个控制信号
Cu cu(reset,zero,negative,inst,pc_ena,rf_wena,dm_ena,dm_wena,dm_wsel,hi_wena,
lo_wena,aluc,sign16,div_sign,mul_sign,m_pc,m_alua,m_alub,m_hi,m_lo,m_rfwa,m_
rfwd);
endmodule
```

(3) 数据存储器模块 dmem. v。

```
module dmem (
    input clk,              //存储器时钟信号,上升沿时向 RAM 内部写入数据
    input ena,              //存储器有效信号,高电平时存储器才运行,否则输出 z
    input wena,             //存储器写有效信号,高电平有效,与 ena 同时有效时才可写存储器
    input [3:0] byteEna,    //字节使能,控制宽度
    input [31:0] addr,      //输入地址,指定数据读写的地址
    input [31:0] data_in,   //存储器写入的数据,在 clk 上升沿时被写入
    output [31:0] data_out //存储器读出的数据
);
```

(4) 寄存器模块 regi. v,异步置位的 D 触发器。

```
module regi(
    input clk,
    input rst,
    input wena,
    input [31:0]idata,
    output [31:0]odata
);
```

(5) 控制单元模块 cu. v。

```
module cu(
    input reset,
```

```
    input zero,
    input negative,
    input [31:0] inst,
    output pc_ena,              //pc 寄存器写信号
    output rf_wena,             //通用寄存器写信号
    output dm_ena,              //数据存储器写信号
    output [3:0]dm_wena,        //数据存储器输入数据选择信号
    output hi_wena,             //hi 寄存器写信号
    output lo_wena,             //lo 寄存器写信号
    output [3:0]aluc,           //alu 执行选择信号
    output sign16,              //有符号数拓展信号
    output div_sign,            //除操作信号
    output mul_sign,            //乘操作信号
    output [2:0]m_pc,           //pc 输入选择信号
    output m_alua,              //alu A 输入端选择信号
    output [1:0]m_alub,         //alu B 输入端选择信号
    output [1:0]m_hi,           //hi 寄存器输入选择信号
    output [1:0]m_lo,           //lo 寄存器输入选择信号
    output [1:0]m_rfwa,         //通用寄存器组 wa 端选择信号
    output [3:0]m_rfwd          //通用寄存器组 wd 端选择信号
);
```

(6) 寄存器组模块 regfile. v。

```
module regfile(
    input clk,                  //寄存器组时钟信号,下降沿写入数据
    input rst,                  //reset 信号,异步复位,高电平时全部寄存器置零
    input we,                   //寄存器读写有效信号,高电平时允许寄存器写入数据,低
                                //电平时允许寄存器读出数据
    input [4:0] waddr,          //写寄存器的地址
    input [31:0] wdata,         //写寄存器数据,数据在 clk 下降沿时被写入
    input [4:0] raddr1,         //所需读取的寄存器的地址
    input [4:0] raddr2,         //所需读取的寄存器的地址
    output [31:0] rdata1,       //raddr1 对应寄存器的输出数据
    output [31:0] rdata2        //raddr2 对应寄存器的输出数据
);
```

(7) alu 模块 alu. v。

```
module alu(
    input [31:0] a,             //32 位输入,操作数 a
    input [31:0] b,             //32 位输入,操作数 b
    input [3:0] aluc,           //4 位输入,控制 alu 的操作
    output reg [31:0] r,        //32 位输出,由 a、b 经过 aluc 指定的操作生成
    output zero,                //0 标志位
    output carry,               //进位标志位
    output negative,            //负数标志位
```

```
    output overflow                 //溢出标志位
);
```

(8) 除法器模块 div. v。

```
module div(
    input clk,
    input reset,
    input start,
    input [31:0]dividend,          //32 位输入,被除数
    input [31:0]divisor,           //32 位输入,除数
    input sign,
    output [31:0]q,                //32 位输出,商
    output [31:0]r,                //32 位输出,余数
    output reg busy                //输出再计算信号
);
```

(9) 乘法器模块 mul. v。

```
module mul(
    input clk,
    input reset,
    input start,
    input [31:0]a,                 //32 位输入,操作数 a
    input [31:0]b,                 //32 位输入,操作数 b
    input sign,
    output [31:0]hi,
    output [31:0]lo,
    output reg busy
);
```

(10) 协处理器模块 cp. v。

```
module cp(
    input clk,
    input reset,
    input [31:0]pc,
    input [31:0]inst,
    input [4:0]addr,
    input [31:0]wdata,
    output [31:0]status,
    output [31:0]epc,
    output [31:0]rdata
);
wire [31:0]cause;
//协处理器行为逻辑模块
cp0 cp0 (clk, reset, mfc0, addr, mtc0, wcau, wsta, wepc, wdata, exc, pc, cause, rdata,
status,epc);
```

```
//协处理器控制信号模块
cu0 cu0(inst,addr,status,mfc0,mtc0,wcau,wsta,wepc,exc,cause);
endmodule
```

(11) 0 号协处理器模块 cp0.v。

```
module cp0(
    input clk,
    input rst,
    input mfc0,
    input [4:0]addr,
    input mtc0,
    input wcau,
    input wsta,
    input wepc,
    input [31:0]data,
    input exc,
    input [31:0]pc,
    input [31:0]cause,
    output [31:0]cp0_out,
    output [31:0]status_out,
    output [31:0]epc_out
);

wire [31:0]cause_in,cause_out,status_in,epc_in,shift_out;

mux mux_cau(mtc0,cause,data,cause_in);
mux mux_sta_shift(exc,status_out>>5,status_out<<5,shift_out);
mux mux_sta(mtc0,shift_out,data,status_in);
mux mux_epc(mtc0,pc,data,epc_in);

regi Cause(clk,rst,wcau,cause_in,cause_out);
regi Status(clk,rst,wsta,status_in,status_out);
regi Epc(clk,rst,wepc,epc_in,epc_out);

mux_cp0_out mux_cp0_out(addr,mfc0,cause_out,status_out,epc_out,cp0_out);

endmodule
```

(12) 0 号协处理器控制器模块 cu0.v。

```
module cu0(
    input [31:0]inst,
    input [4:0]addr,
    input [31:0]status,
    output reg mfc0,
```

```
    output reg mtc0,
    output reg wcau,
    output reg wsta,
    output reg wepc,
    output reg exc,
    output reg [31:0]cause
);
```

第 4 章　简单的流水线 CPU 设计

4.1　实 验 内 容

指令集设计与五级流水线的实现。

4.2　实 现 目 标

最终目标实现的 CPU 是一个五级流水线的精简版 CPU，包括 IF(取指令)、ID(解码)、EX(执行)、MEM(内存操作)、WB(回写)。涉及的结构设计有

- 指令集：RISC。
- 指令集大小：2^5。
- 数据宽度：16bit。
- 数据内存：$2^8 \times 16$bit。
- 指令内存：$2^8 \times 16$bit。
- 通用寄存器：8×16bit。
- 标志寄存器：NF(negative flag)、ZF(zero flag)、CF(carry flag)。
- 控制信号：clock、reset、enable、start。

4.3　流水线 CPU 介绍与设计

4.3.1　CPU 顶层视图

第一步只要求实现简单的五级流水线，不要求实现指令内存、数据内存模块，因此 CPU 内部与内存有关的信号都简化为输入输出信号了，CPU 的顶层视图看起来应该如图 4.1 所示，其中 select_y、y 信号与 CPU 板级测试有关，这一步暂且没用到。

4.3.2　指令集

指令为三地址格式，操作码长度为 5bit，根据操作数的不同，可以把指令分为 3 种类型，即寄存器(R)类型、立即数(I)类型、混合(RI)类型。不过，后面在编写代码时，为了方便，会依据其他标准进行划分。指令格式如图 4.2 所示。

规范一下表示方式，r1 或者 gr[r1]表示访问寄存器 r1，m[r2＋val3]表示访问 r2＋val3 地址，{val2，val3}表示立即数访问，val2 为 MSB，val3 为 LSB。机器代码示例如图 4.3 所示。

为了便于实现，一共列出 28 条机器指令，见表 4.1，剩下未用的 4 个操作码(10100，10101，10110，10111)可自行补充为其他操作，如自增 INC、自减 DEC。下面是指令集的具体格式与操作。

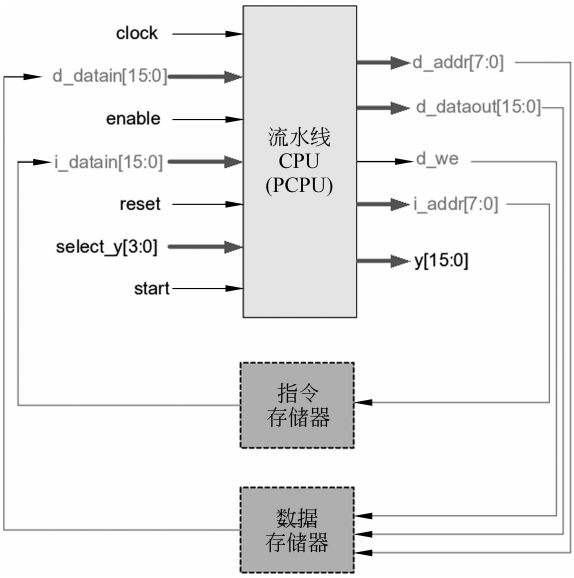


图 4.1　CPU 的顶层视图

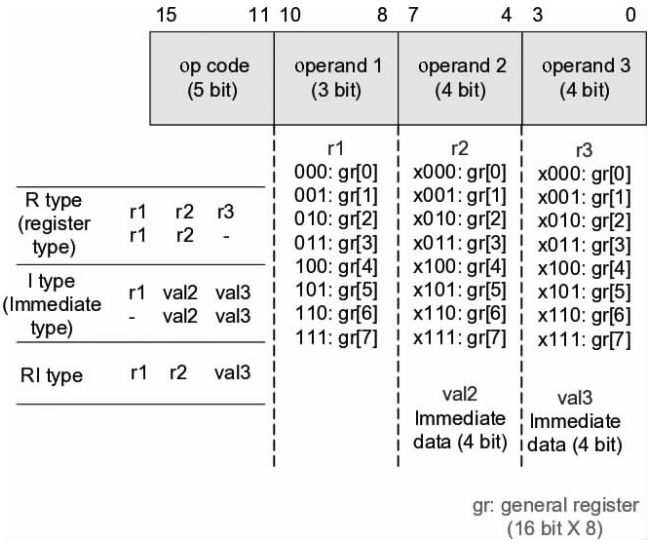


图 4.2　指令格式

```
• LOAD gr1, gr0, 0
• LOAD gr2,  gr0, 1
• NOP
• NOP
• NOP
• ADD gr3, gr1,  gr2
• NOP
• NOP
• NOP
• STORE gr3, gr0, 2
• HALT
```

```
In C code:
Y =  A + B
M[gr0+2] =M[gr0+1]+M[gr0+0]
```

图 4.3　机器代码示例

表 4.1　指令操作码

mnemonie	operand 1	operand 2	operand 3	op code	operation
NOP				#00000	no operation
HALT				#00001	halt
LOAD	r1	r2	val3	#00010	gr[r1]<－m[r2＋val3]
STORE	r1	r2	val3	#00011	m[r2＋val3]<－r1
LDIH	r1	val2	val3	#10000	r1<－r1＋{val2, val3, 0000_0000}
ADD	r1	r2	r3	#01000	r1<－r2＋r3
ADDI	r1	val2	val3	#01001	r1<－r1＋{val2,val3}
ADDC	r1	r2	r3	#10001	r1<－r2＋r3＋CF
SUIH	r1	val2	val3	#10011	r1<－r1－{val2, val3, 0000_0000}
SUB	r1	r2	r3	#01010	r1<－r2－r3
SUBI	r1	val2	val3	#01011	r1<－r1－{val2, val3}
SUBC	r1	r2	r3	#10010	r1<－r2－r3－CF
CMP		r2	r3	#01100	r2－r3, set CF, ZF, NF
AND	r1	r2	r3	#01101	r1<－r2 and r3
OR	r1	r2	r3	#01110	r1<－r2 or r3
XOR	r1	r2	r3	#01111	r1<－r2 xor r3
SLL	r1	r2	val3	#00100	r1 <－r2 shift left logical (val3 bit shift)
SRL	r1	r2	val3	#00110	r1<－r2 shift right logical (val3 bit shift)
SLA	r1	r2	val3	#00101	r1 <－r2 shift left arithmetical (val3 bit shift)
SRA	r1	r2	val3	#00111	r1 <－r2 shift right arithmetical (val3 bit shift)
JUMP		val2	val3	#11000	jump to {val2, val3}
JMPR	r1	val2	val3	#11001	jump to r1 ＋{val2, val3}
BZ	r1	val2	val3	#11010	if ZF＝1 branch to r1＋{val2, val3}
BNZ	r1	val2	val3	#11011	if ZF ＝O branch to r1 ＋{val2, val3}
BN	r1	val2	val3	#11100	if NF＝1 branch to r1＋{val2, val3}
BNN	r1	val2	val3	#11101	if NF ＝O branch to r1＋{val2, val3}
BC	r1	val2	val3	#11110	if CF＝1 branch to r1＋{val2, val3}
BNC	r1	val2	val3	#11111	if CF＝O branch to r1＋{val2, val3}

4.3.3　五级流水线

除指令集外，设计 CPU 最重要的是下面这张 CPU 块级电路图，如图 4.4 所示。五级流水线的代码实现都必须依赖于这张图，因此需要理解图 4.4 中每一步的作用。

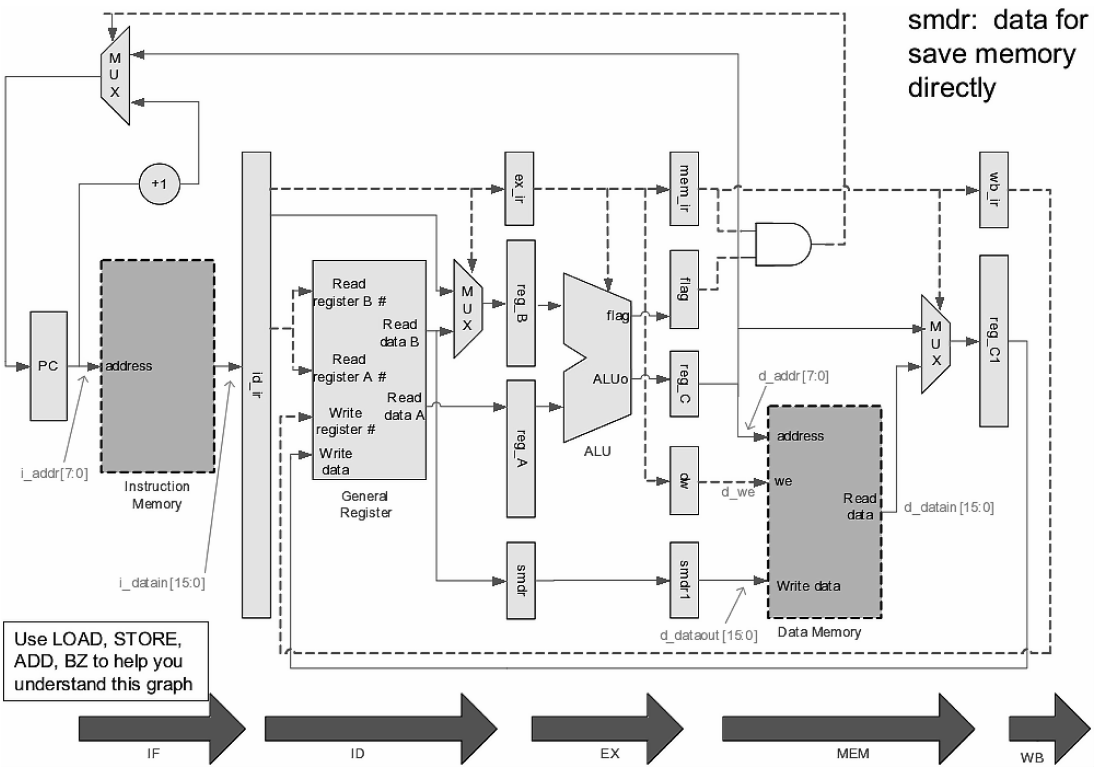


图 4.4　CPU 块级电路图

其实图 4.4 并不复杂，CPU 无非就是组合逻辑电路和时序电路的结合，而图中所有矩形框标出的都是 CPU 内部寄存器，整个电路图展示了 CPU 内部指令以及数据的流动方向。每到时钟上升沿，上一级流水线的寄存器的数据就会经过中间的组合逻辑电路流动到下一级流水线的寄存器，因此，5 个时钟周期之后一条机器指令便执行完毕了。简单描述一条指令的执行过程就是，首先根据 PC 的值到内存中取一条指令，解码指令提取两个操作数进行运算，根据指令功能以及运算结果决定是否访问数据内存以及如何访问，最后根据指令功能决定是否进行回写操作，即修改寄存器的值。

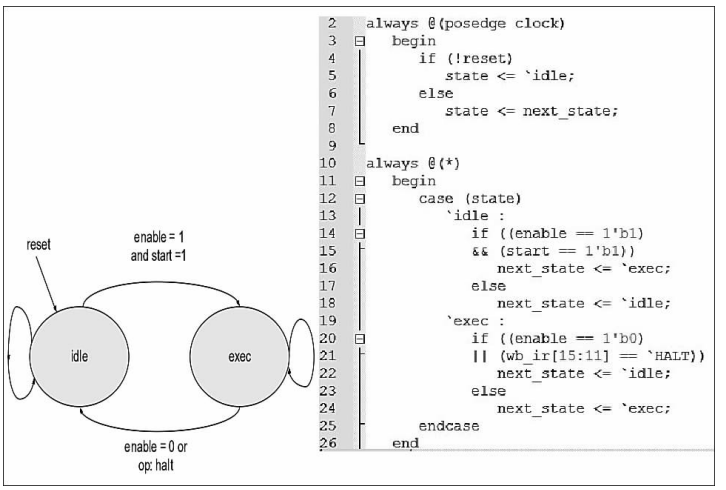
下面分别讲解 CPU 控制以及五级流水线每一级的行为，为了简单起见，这里仅考虑 NOP、HALT、LOAD、STORE、ADD、CMP、BZ、BN 这 8 条指令，明白流水线的行为之后，再加上其他指令道理也一样。

1. CPU 控制

CPU 控制基于状态机，只有 idle 和 exec 两个状态，CPU 在 idle 状态下只有 enable、start 同时使能，才会进入 exec 状态。CPU 控制如图 4.5 所示。

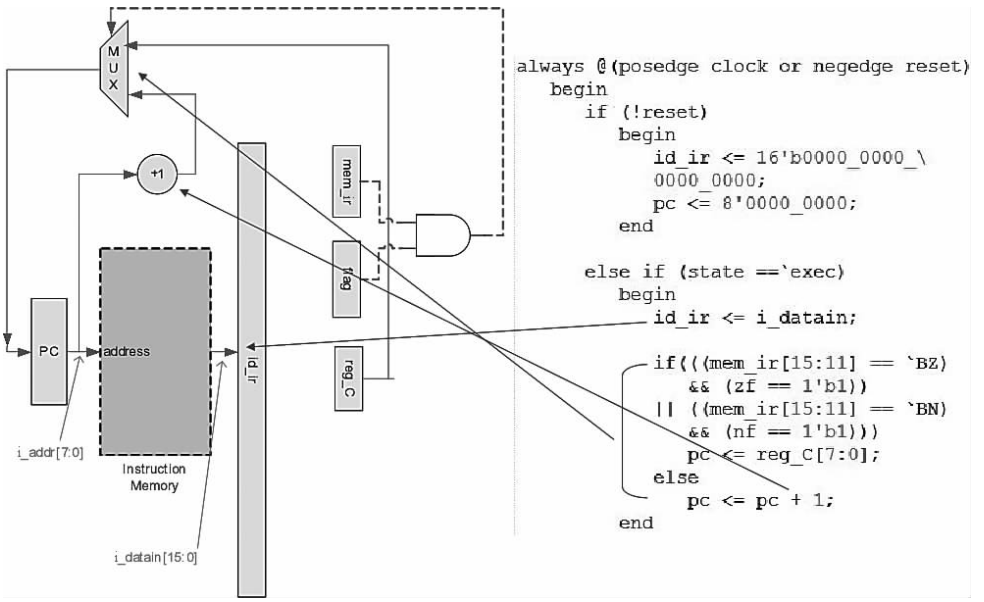
2. 取指令 IF 级的电路

IF 阶段的任务是根据 PC 的值从指令内存中读取一条指令，并且设置下一周期 PC 的值(指令可以顺序执行，也可以跳转到某个特定的地址执行)。因为读取内存是内存模块实现的功能，因此这里 CPU 只需要给出指令地址 i_addr，就能得到对应的指令 i_datain。IF 阶段数据通路如图 4.6 所示。




```
2   always @(posedge clock)
3     begin
4       if (!reset)
5         state <= `idle;
6       else
7         state <= next_state;
8     end
9
10  always @(*)
11    begin
12      case (state)
13        `idle :
14          if ((enable == 1'b1)
15          && (start == 1'b1))
16            next_state <= `exec;
17          else
18            next_state <= `idle;
19        `exec :
20          if ((enable == 1'b0)
21          || (wb_ir[15:11] == `HALT))
22            next_state <= `idle;
23          else
24            next_state <= `exec;
25      endcase
26    end
```

图 4.5　CPU 控制




```
always @(posedge clock or negedge reset)
   begin
      if (!reset)
         begin
            id_ir <= 16'b0000_0000_\
            0000_0000;
            pc <= 8'0000_0000;
         end

      else if (state ==`exec)
         begin
            id_ir <= i_datain;

            if((mem_ir[15:11] == `BZ)
               && (zf == 1'b1))
            || ((mem_ir[15:11] == `BN)
               && (nf == 1'b1)))
               pc <= reg_C[7:0];
            else
               pc <= pc + 1;
         end
```

图 4.6　IF 阶段数据通路

3. 指令译码 ID 级的电路

ID 阶段要根据指令的功能(即操作码)从指令中提取对应的操作数,操作数可能来自通用寄存器 r0～r7,也可能是立即数。另外,如果指令是 STORE 指令,也要准备好存储到内存中的数据。ID 阶段数据通路如图 4.7 所示。

4. 指令执行 EX 级的电路

EX 阶段执行的是 ALU 运算和标志寄存器设置。另外,如果是 STORE 指令,也要给出内存写的使能信号 dw 以及将要写到内存中的数据 smdr1。EX 阶段数据通路如图 4.8 所示。

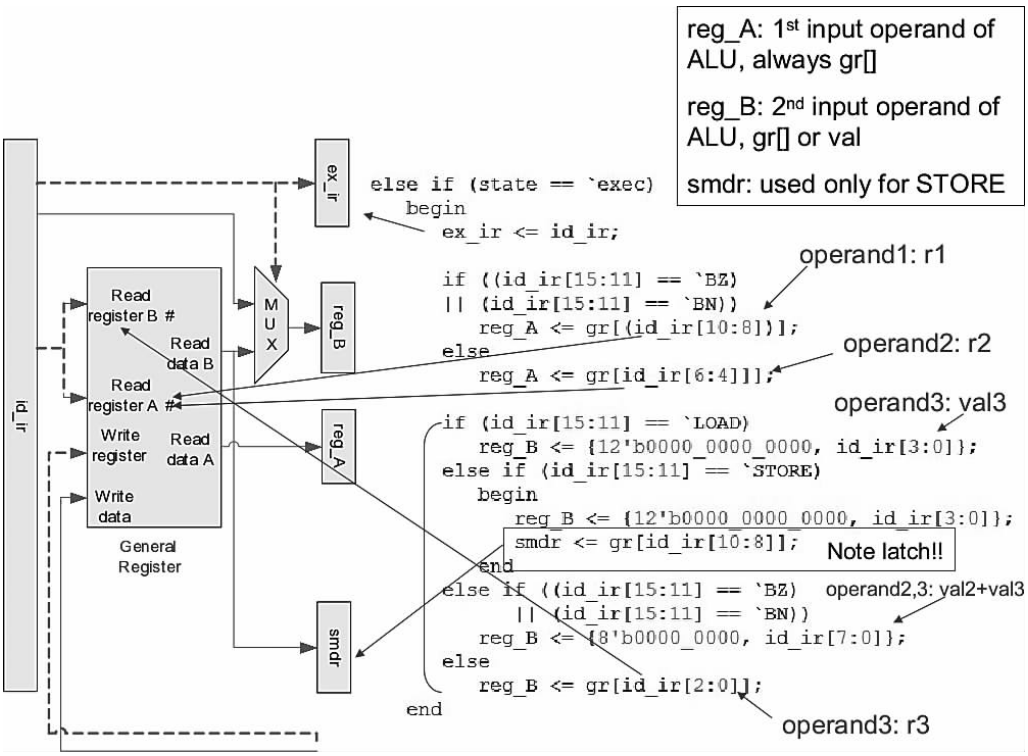


图 4.7　ID 阶段数据通路

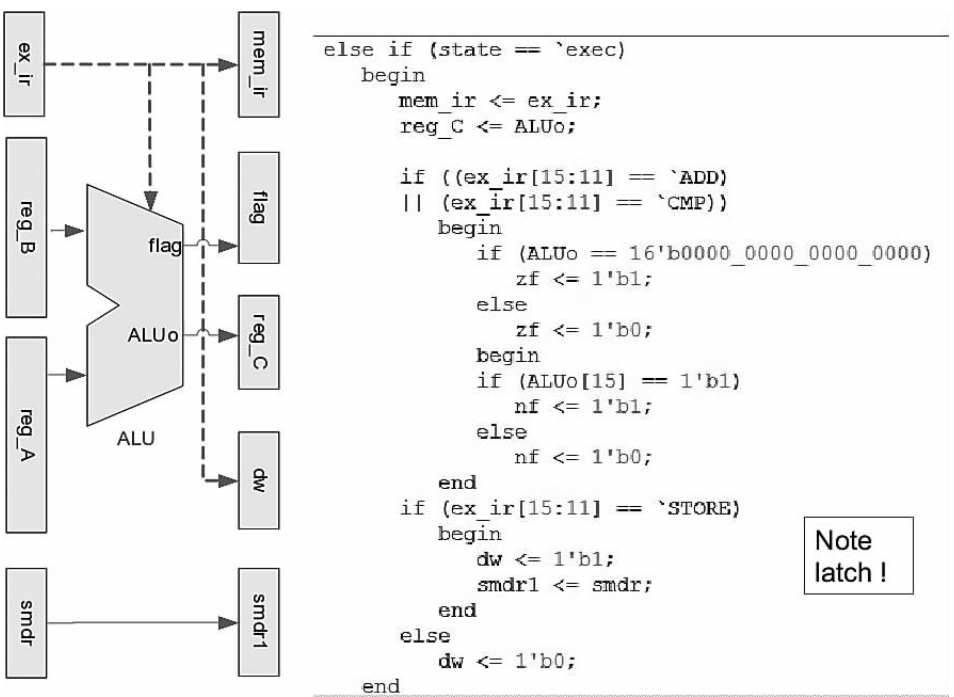


图 4.8　EX 阶段数据通路

5. 存储器访问 MEM 级的电路

MEM 阶段要根据指令功能和上一阶段的运算结果(内存操作的时候作为内存地址)决定是否要访问内存以及如何访问,只对需要内存操作的指令有效。MEM 阶段数据通路如图 4.9 所示。

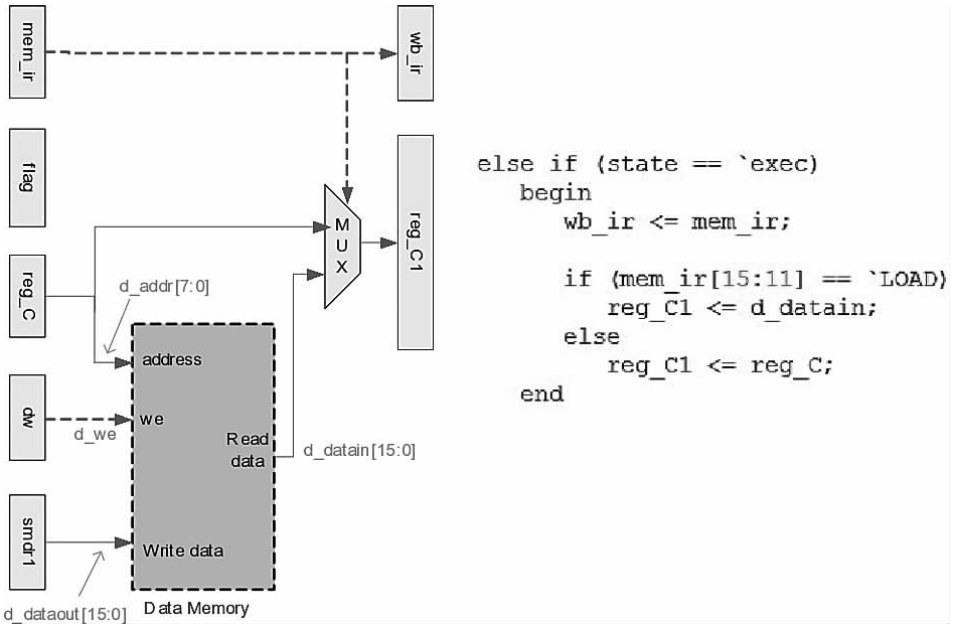


图 4.9　MEM 阶段数据通路

6. 写回指令 WB 级的电路

WB 阶段同样根据指令的功能以及上一阶段的结果决定是否要修改寄存器的值以及如何修改,只对需要修改寄存器值的指令有效。WB 阶段数据通路如图 4.10 所示。

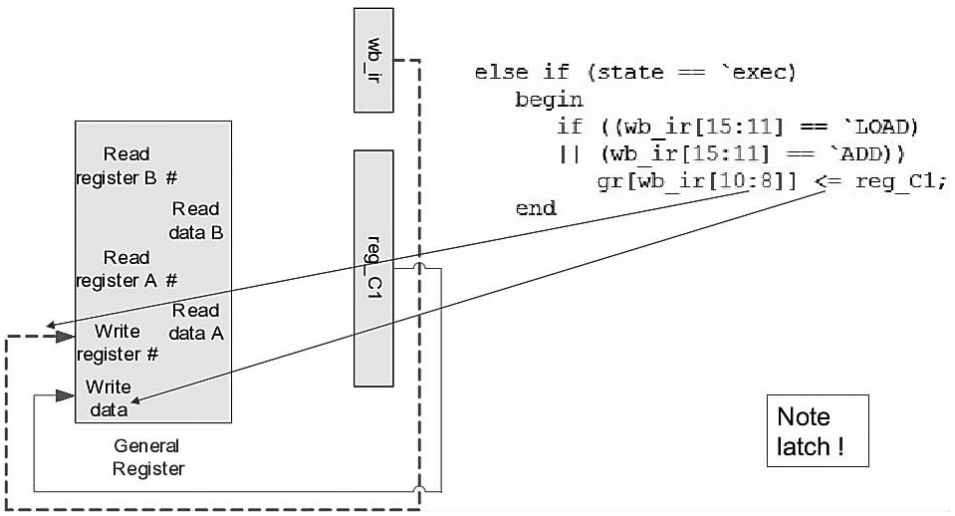


图 4.10　WB 阶段数据通路

7. 代码实现

上面的设计中只给出了时序电路的部分代码，而且只针对个别指令，另外还应该有一部分组合逻辑电路的代码是处理上一级流水线寄存器的数据如何流动到下一级寄存器的。下面是完整的代码实现。

程序 4.1　def.v

```
`define idle      1'b0
`define exec      1'b1

`define NOP       5'b0_0000
`define HALT      5'b0_0001
`define LOAD      5'b0_0010
`define STORE     5'b0_0011
`define SLL       5'b0_0100
`define SLA       5'b0_0101
`define SRL       5'b0_0110
`define SRA       5'b0_0111
`define ADD       5'b0_1000
`define ADDI      5'b0_1001
`define SUB       5'b0_1010
`define SUBI      5'b0_1011
`define CMP       5'b0_1100
`define AND       5'b0_1101
`define OR        5'b0_1110
`define XOR       5'b0_1111
`define LDIH      5'b1_0000
`define ADDC      5'b1_0001
`define SUBC      5'b1_0010
`define SUIH      5'b1_0011
`define JUMP      5'b1_1000
`define JMPR      5'b1_1001
`define BZ        5'b1_1010
`define BNZ       5'b1_1011
`define BN        5'b1_1100
`define BNN       5'b1_1101
`define BC        5'b1_1110
`define BNC       5'b1_1111

`define gr0       3'b000
`define gr1       3'b001
`define gr2       3'b010
`define gr3       3'b011
`define gr4       3'b100
`define gr5       3'b101
`define gr6       3'b110
```