移动操作指令的添加

一、指令说明

movn、movz、mfhi、mthi、mflo、mtlo
本次要实现两个特殊寄存器 HI 和 LO。
指令格式如表：

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	rs	rt	rd	00000	MOVN 001011	movn指令
SPECIAL 000000	rs	rt	rd	00000	MOVZ 001010	movz指令
SPECIAL 000000	00000	00000	rd	00000	MFHI 010000	mfhi指令
SPECIAL 000000	00000	00000	rd	00000	MFLO 010010	mflo指令
SPECIAL 000000	rs	00000	00000	00000	MTHI 010001	mthi指令
SPECIAL 000000	rs	00000	00000	00000	MTLO 010011	mtlo指令

图 6-1　移动操作指令的格式

后四条指令需要读取 HILO 寄存器，设计在执行阶段才能读取到。

步骤：
（1）　在译码阶段依据指令，给出运算类型 alusel_o、运算子类型 aluop_o 的值；
（2）　在执行阶段获取 HI 或 LO 寄存器的值，作为要写入或要获取的寄存器的数据，并将这些信息传递到访存阶段；
（3）　访存阶段将这些信息在传递到回写阶段；
（4）　回写阶段依据这些信息修改目的寄存器。

我们在回写阶段增加 HILO 寄存器模块，如图：

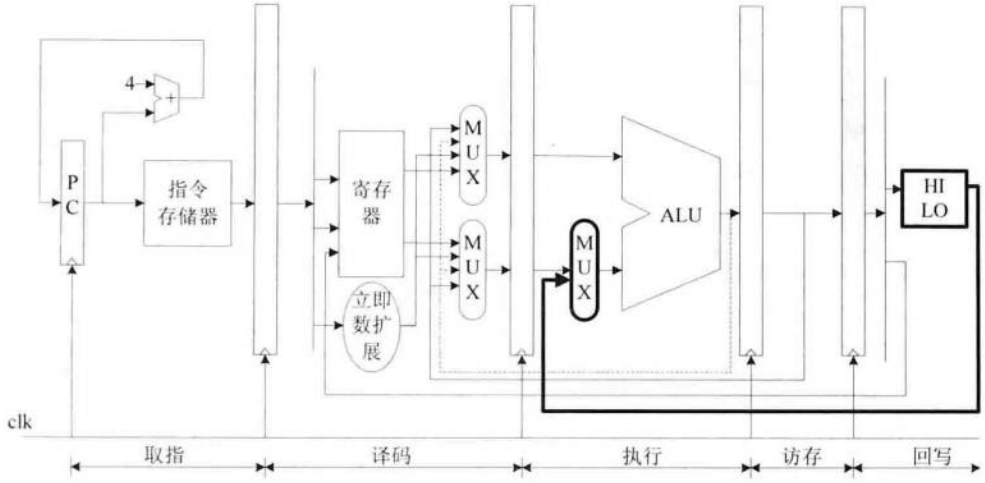

图 6-2　添加移动操作指令后的数据流图

二、出现新的数据相关

考虑 HILO 寄存器的处理过程，这两条指令会在执行阶段读取 HILO 的值，如果直接使用其里面的值可能是不正确的，如下图：

```
1.    lui $1,0x0000          # $1 = 0x00000000
2.    lui $2,0xffff          # $1 = 0xffff0000
3.    mthi $0                # hi = 0x00000000
4.    mthi $1                # hi = 0x00000000
5.    mthi $2                # hi = 0xffff0000
6.    mfhi $4                # $4 = 0xffff0000
```

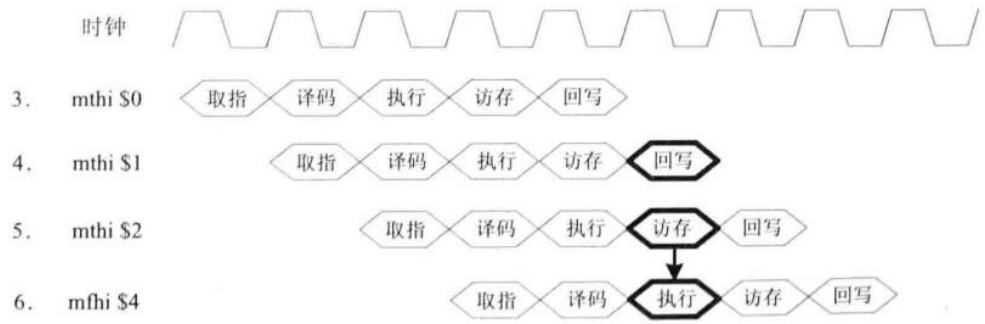

图 6-3　HI、LO 寄存器带来的数据相关示意图

我们发现，在第六步的执行阶段读取到 HILO 寄存器的值时，第四步才刚刚回写，第五步还没有回写，所以获取到了是第四步回写进 HILO 的值，而正确的应该是获取到第五步中回写进 HILO 的值。这便产生了数据相关，解决办法还是数据前推。

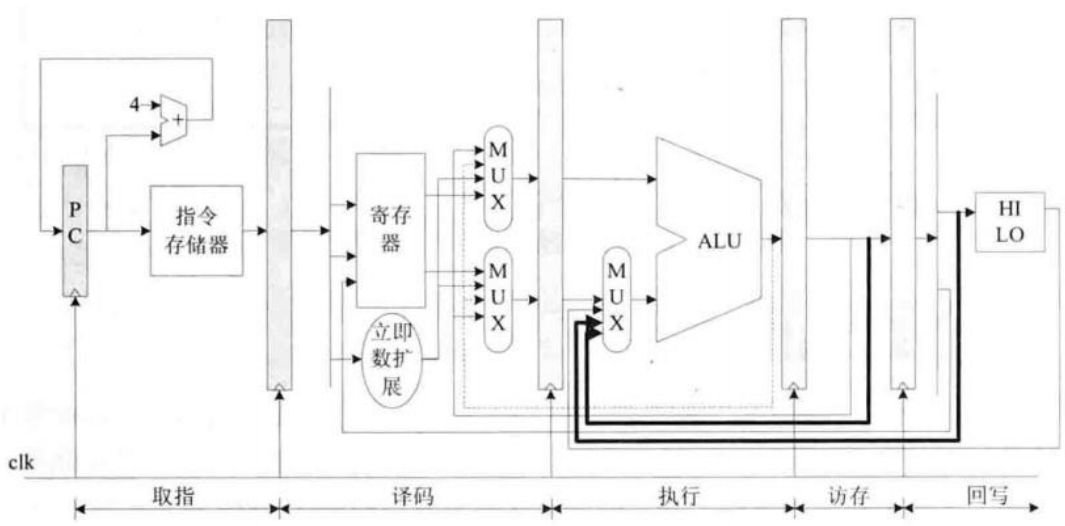

图 6-4　解决 HI、LO 寄存器带来的数据相关问题后的数据流图

因此我们进行系统结构的修改，如图：

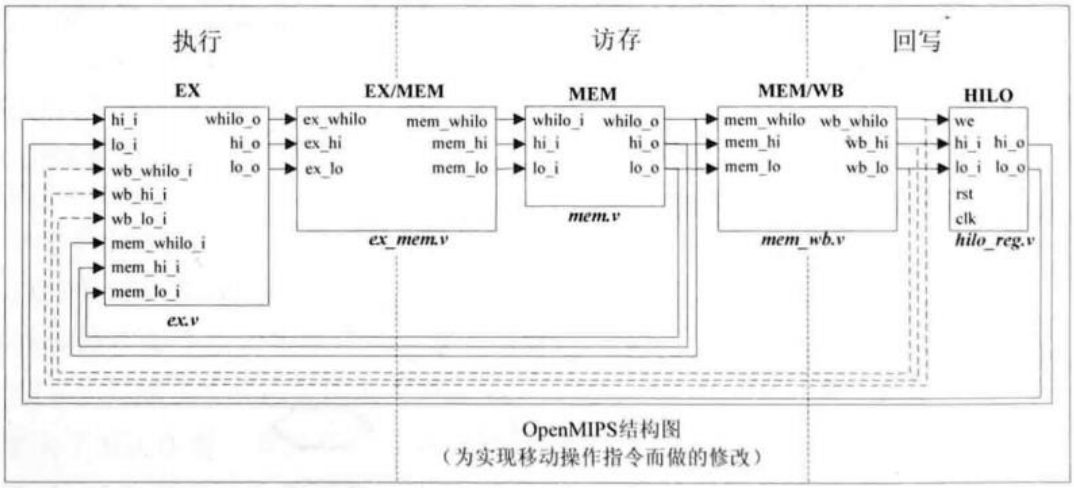

图 6-5　为实现移动操作指令而对 OpenMIPS 系统结构所做的修改

主要体现在三个方面：

（1）　增加了 HILO 模块，用于实现 HILO 寄存器；

（2）　执行阶段 EX 模块增加 while_o、hi_o、lo_o 接口，表示是否要写入 HILO、要写入 HI 的值和要写入 LO 的值。这三个接口传递出来的信息会通过后面的模块一直传递到回写模块，最终传递到 HILO 模块；

（3）　执行阶段的 EX 模块增加了与 HILO 寄存器有关的输入接口，包括为解决 HILO 寄存器的数据相关问题而引入的接口。

三、HI、LO 寄存器的实现

接口描述如表：

表 6-1　HILO 模块的接口

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	we	1	输入	HI、LO 寄存器写使能信号
4	hi_i	32	输入	要写入 HI 寄存器的值
5	lo_i	32	输入	要写入 LO 寄存器的值
6	hi_o	32	输出	HI 寄存器的值
7	lo_o	32	输出	LO 寄存器的值

源代码 **hilo_reg.v**：

```
module hilo_reg(
    input wire clk,
    input wire rst,

    //Input port
    input wire we,
    input wire [`RegBus] hi_i,
    input wire [`RegBus] lo_i,

    //Output port
    output reg [`RegBus] hi_o,
    output reg [`RegBus] lo_o
    );

always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        hi_o <= `ZeroWord;
        lo_o <= `ZeroWord;
    end else if ((we == `WriteEnable)) begin
        hi_o <= hi_i;
        lo_o <= lo_i;
    end
end
endmodule
```

很简单，没什么说的。

四、修改译码阶段的 ID 模块

相关指令的功能码对照：

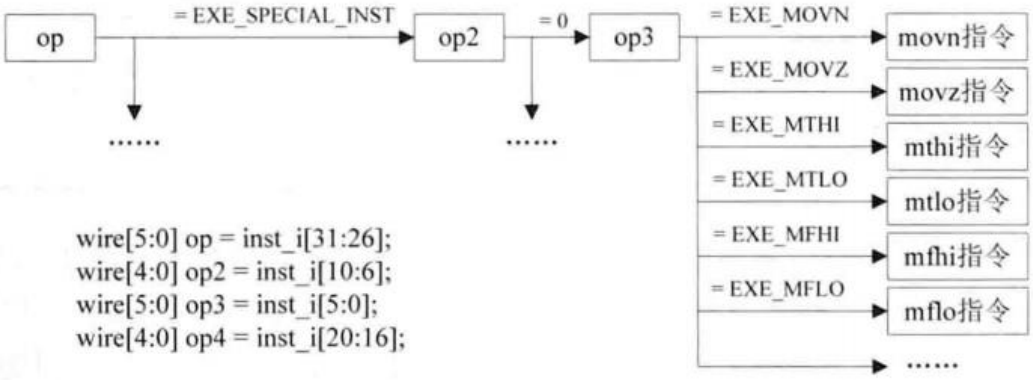

图 6-6　确定移动操作指令的过程

根据 op3 的不同进行 ID 模块的修改：

源代码 **define.v**：

```
`define EXE_RES_MOVE            3'b011

`define EXE_MOVZ_OP             8'b00001010
`define EXE_MOVN_OP             8'b00001011
`define EXE_MFHI_OP             8'b00010000
`define EXE_MTHI_OP             8'b00010001
`define EXE_MFLO_OP             8'b00010010
`define EXE_MTLO_OP             8'b00010011

`define EXE_MOVZ                6'b001010
`define EXE_MOVN                6'b001011
`define EXE_MFHI                6'b010000
`define EXE_MTHI                6'b010001
`define EXE_MFLO                6'b010010
`define EXE_MTLO                6'b010011
```

源代码 **id.v**：

```
`EXE_MFHI: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `EXE_MFHI_OP;
```

```
        alusel_o <= `EXE_RES_MOVE;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        instvalid <= `InstValid;
        end
    `EXE_MFLO: begin
        wreg_o <= `WriteEnable;
        aluop_o <= `EXE_MFLO_OP;
        alusel_o <= `EXE_RES_MOVE;
        reg1_read_o <= 1'b0;
        reg2_read_o <= 1'b0;
        instvalid <= `InstValid;
    end
    `EXE_MTHI: begin
        wreg_o <= `WriteDisable;
        aluop_o <= `EXE_MTHI_OP;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        instvalid <= `InstValid;
    end
    `EXE_MTLO: begin
        wreg_o <= `WriteDisable;
        aluop_o <= `EXE_MTLO_OP;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b0;
        instvalid <= `InstValid;
    end
    `EXE_MOVN: begin
        aluop_o <= `EXE_MOVN_OP;
        alusel_o <= `EXE_RES_MOVE;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b1;
        instvalid <= `InstValid;
        if (reg2_o != `ZeroWord) begin
            wreg_o <= `WriteEnable;
        end else begin
            wreg_o <= `WriteDisable;
        end
    end
    `EXE_MOVZ: begin
        aluop_o <= `EXE_MOVZ_OP;
        alusel_o <= `EXE_RES_MOVE;
        reg1_read_o <= 1'b1;
        reg2_read_o <= 1'b1;
```

```
        instvalid <= `InstValid;
        if (reg2_o != `ZeroWord) begin
            wreg_o <= `WriteEnable;
        end else begin
            wreg_o <= `WriteDisable;
        end
    end
```

值得注意的是：除了 mthi、mtlo 外的四条移动指令的运算类型 alusel_o 都是 EXE_RES_MOVE。

另外，movz 指令的译码过程需要读取 rs 和 rt 寄存器的值，所以 reg1_read_o 和 reg2_read_o 都是 1，默认通过 RegFile 模块读端口 1 读取的寄存器地址 reg1_addr_o 的值是指令的 21~25bit，正是 movz 指令中的 rs，默认通过 RegFile 模块及短期地址 reg2_addr_o 的值是指令的 16~20bit，正是 movz 指令中的 rt。所以如果 reg2_o 为 0，则 wreg_o 为 WriteEnable。

五．修改执行阶段的 EX 模块

译码阶段的结果会传递到执行阶段，执行阶段据此进行计算，考虑到执行阶段需要读写 HILO 寄存器，还要解决数据相关问题，我们增加下表所示的接口：

表 6-2　EX 模块要增加的接口

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	hi_i	32	输入	HILO 模块给出的 HI 寄存器的值
2	lo_i	32	输入	HILO 模块给出的 LO 寄存器的值
3	mem_whilo_i	1	输入	处于访存阶段的指令是否要写 HI、LO 寄存器
4	mem_hi_i	32	输入	处于访存阶段的指令要写入 HI 寄存器的值
5	mem_lo_i	32	输入	处于访存阶段的指令要写入 LO 寄存器的值
6	wb_whilo_i	1	输入	处于回写阶段的指令是否要写 HI、LO 寄存器
7	wb_hi_i	32	输入	处于回写阶段的指令要写入 HI 寄存器的值
8	wb_lo_i	32	输入	处于回写阶段的指令要写入 LO 寄存器的值
9	whilo_o	1	输出	执行阶段的指令是否要写 HI、LO 寄存器
10	hi_o	32	输出	执行阶段的指令要写入 HI 寄存器的值
11	lo_o	32	输出	执行阶段的指令要写入 LO 寄存器的值

这些接口便实现了数据前推，避免了数据相关问题。

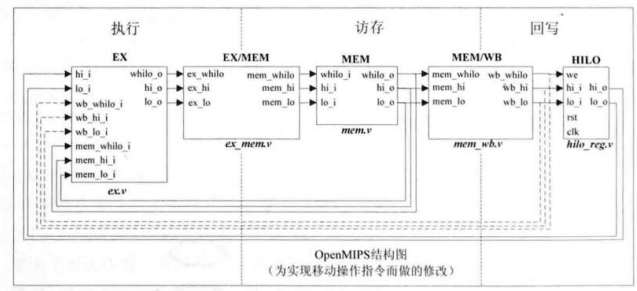

图 6-5　为实现移动操作指令而对 OpenMIPS 系统结构所做的修改

源代码 **ex.v：**

```
//The value of HILO    form HILO register
    input wire [`RegBus] hi_i,
    input wire [`RegBus] lo_i,

    //The value from WB stage
    input wire [`RegBus] wb_hi_i,
    input wire [`RegBus] wb_lo_i,
    input wire wb_whilo_i,

    //The value from MEM stagee
    input wire [`RegBus] mem_hi_i,
    input wire [`RegBus] mem_lo_i,
    input wire mem_whilo_i,

    //Write operation to HILO register
    output reg [`RegBus] hi_o,
    output reg [`RegBus] lo_o,
    output reg whilo_o,

//Save the result of movement calculation
    reg [`RegBus] moveres;

    reg [`RegBus] HI;
    reg [`RegBus] LO;

//Get the newest value of HILO
    always @ (*) begin
        if (rst == `RstEnable) begin
            {HI, LO} <= {`ZeroWord, `ZeroWord};
        end else if (mem_whilo_i == `WriteEnable) begin
            {HI, LO} <= {mem_hi_i, mem_lo_i};
        end else if (wb_whilo_i == `WriteEnable) begin
            {HI, LO} <= {wb_hi_i, wb_lo_i};
        end else begin
```

```
                {HI, LO} <= {hi_i, lo_i};
            end
        end

//MOVES
        ys @(*) begin
        if (rst == `RstEnable) begin
            moveres <= `ZeroWord;
        end else begin
            case (aluop_i)
                `EXE_MFHI_OP: begin
                    moveres <= HI;
                end
                `EXE_MFLO_OP: begin
                    moveres <= LO;
                end
                `EXE_MOVN_OP: begin
                    moveres <= reg1_i;
                end
                `EXE_MOVZ_OP: begin
                    moveres <= reg1_i;
                end
            endcase
        end
end

`EXE_RES_MOVE: begin
        wdata_o <= moveres;
end

always @ (*) begin
        if (rst == `RstEnable) begin
            whilo_o <= `WriteDisable;
            hi_o <= `ZeroWord;
            lo_o <= `ZeroWord;
        end else if (aluop_i == `EXE_MTHI_OP) begin
            whilo_o <= `WriteEnable;
            hi_o <= reg1_i;
            lo_o <= LO;
        end else if (aluop_i == `EXE_MTLO_OP) begin
            whilo_o <= `WriteEnable;
            hi_o <= HI;
            lo_o <= reg1_i;
        end else begin
```

```
            whilo_o <= `WriteDisable;
            hi_o <= `ZeroWord;
            lo_o <= `ZeroWord;
        end
end
```

六、修改 EX/MEM 模块

EX 模块新增加的输出接口 whilo_o、hi_o、lo_o 连接到 EX/MEM 模块，需要增加下列接口：

表 6-3　EX/MEM 模块要增加的接口

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	ex_whilo	1	输入	执行阶段的指令是否要写 HI、LO 寄存器
2	ex_hi	32	输入	执行阶段的指令要写入 HI 寄存器的值
3	ex_lo	32	输入	执行阶段的指令要写入 LO 寄存器的值
4	mem_whilo	1	输出	访存阶段的指令是否要写 HI、LO 寄存器
5	mem_hi	32	输出	访存阶段的指令要写入 HI 寄存器的值
6	mem_lo	32	输出	访存阶段的指令要写入 LO 寄存器的值

源代码 **ex_mem.v**

```
module ex_mem(
    …
    input wire [`RegBus] ex_hi,
    input wire [`RegBus] ex_lo,
    input wire ex_whilo,

    output reg [`RegBus] mem_hi,
    output reg [`RegBus] mem_lo,
    output reg mem_whilo
    );

    always @(posedge clk) begin
        if (rst == `RstEnable) begin
            …
            mem_hi <= `ZeroWord;
            mem_lo <= `ZeroWord;
            mem_whilo <= `WriteDisable;
        end begin
            …
```

```
                mem_hi <= ex_hi;
                mem_lo <= ex_lo;
                mem_whilo <= ex_whilo;
            end
        end
    endmodule
```

很简单没什么说的。

七、修改访存阶段的 MEM 模块

将 MEM 模块新增加的输出接口 mem_hi, mem_lo, mem_whilo 连接到访存阶段的 MEM 模块，需要给 MEM 模块增加如下表所示的接口：

表 6-4 MEM 模块要增加的接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	whilo_i	1	输入	访存阶段的指令是否要写 HI、LO 寄存器
2	hi_i	32	输入	访存阶段的指令要写入 HI 寄存器的值
3	lo_i	32	输入	访存阶段的指令要写入 LO 寄存器的值
4	whilo_o	1	输出	访存阶段的指令最终是否要写 HI、LO 寄存器
5	hi_o	32	输出	访存阶段的指令最终要写入 HI 寄存器的值
6	lo_o	32	输出	访存阶段的指令最终要写入 LO 寄存器的值

源代码 **mem.v**

```
module mem(
    input wire whilo_i,
    input wire [`RegBus] hi_i,
    input wire [`RegBus] lo_i,

    output reg whilo_o,
    output reg [`RegBus] hi_o,
    output reg [`RegBus] lo_o
    );

    always @(*) begin
        if (rst == `RstEnable) begin
            …
            whilo_o <= `WriteDisable;
            hi_o <= `ZeroWord;
            lo_o <= `ZeroWord;
        end else begin
```

```
            …
            whilo_o <= whilo_i;
            hi_o <= hi_i;
            lo_o <= lo_i;
        end
    end

endmodule
```

也很简单，不用解释了。

八、修改 MEM/WB 模块

该模块需要将 MEM 模块新增的输出接口 whilo_o,hi_o,lo_o 连接到自身，需要添加如下表所示的接口：

表 6-5 MEM/WB 模块要增加的接口

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	mem_whilo	1	输入	访存阶段的指令是否要写 HI、LO 寄存器
2	mem_hi	32	输入	访存阶段的指令要写入 HI 寄存器的值
3	mem_lo	32	输入	访存阶段的指令要写入 LO 寄存器的值
4	wb_whilo	1	输出	回写阶段的指令是否要写 HI、LO 寄存器
5	wb_hi	32	输出	回写阶段的指令要写入 HI 寄存器的值
6	wb_lo	32	输出	回写阶段的指令要写入 LO 寄存器的值

源代码 **mem_wb：**

```
module mem_wb(
    input wire [`RegBus] mem_hi,
    input wire [`RegBus] mem_lo,
    input wire mem_whilo,

    output reg [`RegBus] wb_hi,
    output reg [`RegBus] wb_lo,
    output reg wb_whilo
    );

    always @(posedge clk) begin
        if (rst == `RstEnable) begin
            …
            wb_hi <= `ZeroWord;
            wb_lo <= `ZeroWord;
```

```
            wb_whilo <= `WriteDisable;
        end else begin
            …
            wb_hi <= mem_hi;
            wb_lo <= mem_lo;
            wb_whilo <= mem_whilo;
        end
    end
endmodule
```

没什么说的…

九、修改 OpenMIPS 顶层模块

我们在 EX、EX/MEM、MEM、MEM/WB 模块中添加了许多接口，又增加了新的 hilo_reg 模块，我们需要在 OpenMIPS 模块中例化，例化虽然简单，但是变量太多，容易弄混，具体代码如下：

```
wire [`RegBus] ex_hi_o;
wire [`RegBus] ex_lo_o;
wire    ex_whilo_o;

wire [`RegBus] mem_hi_i;
wire [`RegBus] mem_lo_i;
wire mem_whilo_i;

wire [`RegBus] mem_hi_o;
wire [`RegBus] mem_lo_o;
wire mem_whilo_o;

wire [`RegBus] wb_hi_i;
wire [`RegBus] wb_lo_i;
wire wb_whilo_i;

wire [`RegBus] hi;
wire [`RegBus] lo;

ex ex0(
    …
    .hi(hi),
    .lo(lo),
    …
```

```
        .wb_whjilo_i(wb_whilo_i),
        .wb_hi_i(wb_hi_i),
        .wb_lo_i(wb_lo_i),
        .mem_whilo_i(mem_whilo_o),
        .mem_hi_i(mem_hi_o),
        .mem_lo_i(mem_lo_o),
        …
        .whilo_o(ex_whilo_o),
        .hi_o(ex_hi_o),
        .lo_o(ex_lo_o)
);


ex_mem ex_mem0(
        …
        .ex_whilo(ex_whilo_o),
        .ex_hi(ex_hi_o),
        .ex_lo(ex_lo_o),
        …
        .mem_whilo(mem_whilo_i),
        .mem_hi(mem_hi_i),
        .mem_lo(mem_lo_i)
);

mem mem0(
        …
        .whilo_i(mem_whilo_i),
        .hi_i(mem_hi_i),
        .lo_i(mem_lo_i),
        …
        .whilo_o(mem_whilo_o),
        .hi_o(mem_hi_o),
        .lo_o(mem_lo_o)
);

mem_wb mem_wb0(
        …
        .mem_whilo(mem_whilo_o),
        .mem_hi(mem_hi_o),
        .mem_lo(mem_lo_o),
        …
        .wb_whilo(wb_whilo_i),
        .wb_hi(wb_hi_i),
        .wb_lo(wb_lo_i)
```

```
);

hilo_reg hilo_reg0(
        .clk(clk),
        .rst(rst),
        //Input
        .we(wb_whilo_i),
        .hi_i(wb_hi_i),
        .lo_i(wb_lo_i),
        //Output
                .hi_o(hi),
                .lo_o(lo)
);
```

十、测试程序

测试程序如下：

```
    .org 0x0
    .set noat
    .global _start
_start:
    lui $1,0x0000              # $1 = 0x00000000
    lui $2,0xffff              # $2 = 0xffff0000
    lui $3,0x0505              # $3 = 0x05050000
    lui $4,0x0000              # $4 = 0x00000000

    movz $4,$2,$1              # $4 = 0xffff0000
    movn $4,$3,$1              # $4 = 0xffff0000
    movn $4,$3,$2              # $4 = 0x05050000
    movz $4,$2,$3              # $4 = 0x05050000

    mthi $0                    # hi = 0x00000000
    mthi $2                    # hi = 0xffff0000
    mthi $3                    # hi = 0x05050000
    mfhi $4                    # $4 = 0x05050000

    mtlo $3                    # lo = 0x05050000
    mtlo $2                    # lo = 0xffff0000
    mtlo $1                    # lo = 0x00000000
    mflo $4                    # $4 = 0x00000000
```

仿真结果如下：

十一、总结

至此，我们已经完成了基于五级流水 CPU 的近二十条指令了，我们发现只需要在原有框架上进行增加即可，因此一开始的框架非常重要。MIPS 指令集巧妙的将 32 位的指令分为 op、op2、op3、op4 进行判断，巧妙利用分支语句将其分类。在使用 Forwarding 技术时，在每个模块上增加接口以解决数据相关问题。

在一开始的编码中，会有各种各样的问题，比如语法不熟悉，变量太多记不住，记混淆了，接口不知道该怎么命名，该怎么连接和传值，逻辑结构该怎么设计……但是做完了十几条指令后，发现编码并不是最重要的，重要的是 OpenMIPS 的系统结构图，就是那一根根线连起来的电路图，我们所有的代码都是基于那张图来写的，如果你足够熟练，可以照着那张图完成全部的编码。在本章，最重要的一张图如下：

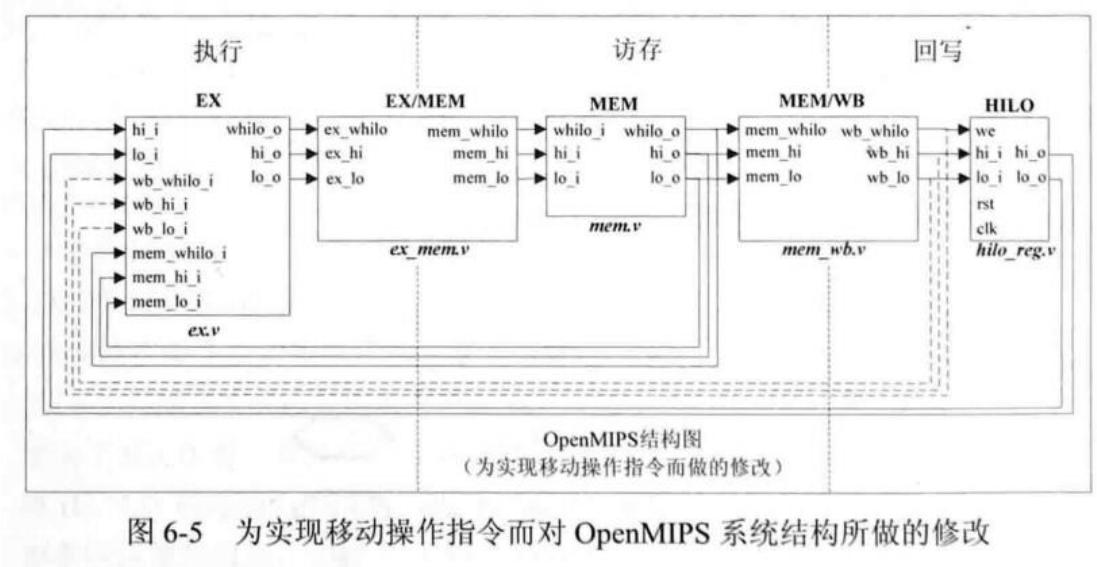

图 6-5　为实现移动操作指令而对 OpenMIPS 系统结构所做的修改

俗话说，万事开头难，当我们理解了其中的原理后，便能得心应手了。

下一章我们还要增加二十几条指令，代码量会越来越大，但是熟练之后，反而没有那么大了。