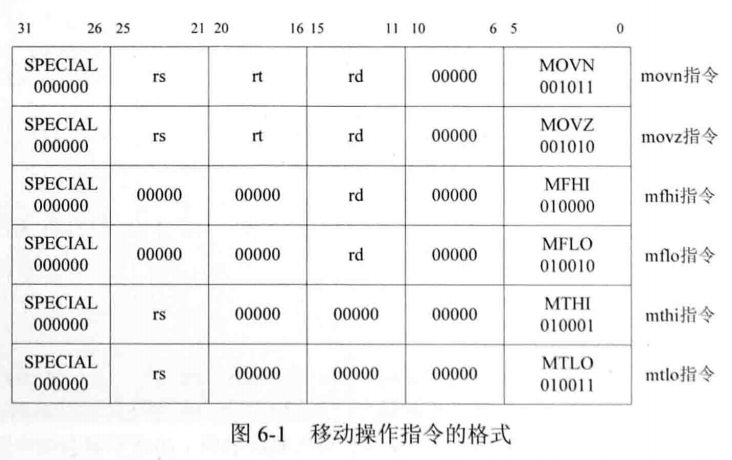
移动操作指令的添加

# 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：

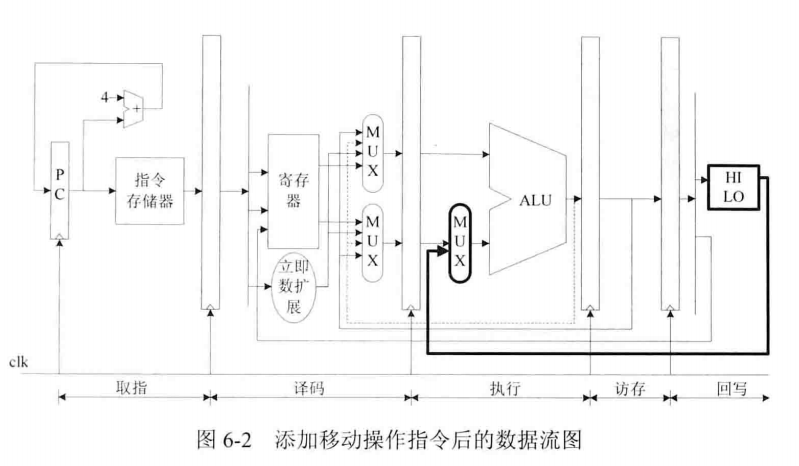


后四条指令需要读取HILO寄存器，设计在执行阶段才能读取到。

步骤：

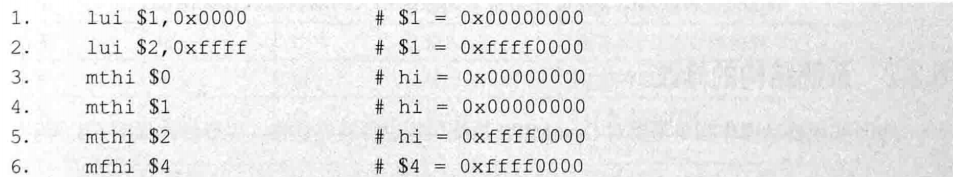
1. 在译码阶段依据指令，给出运算类型alusel\_o、运算子类型aluop\_o的值；
2. 在执行阶段获取HI或LO寄存器的值，作为要写入或要获取的寄存器的数据，并将这些信息传递到访存阶段；
3. 访存阶段将这些信息在传递到回写阶段；
4. 回写阶段依据这些信息修改目的寄存器。

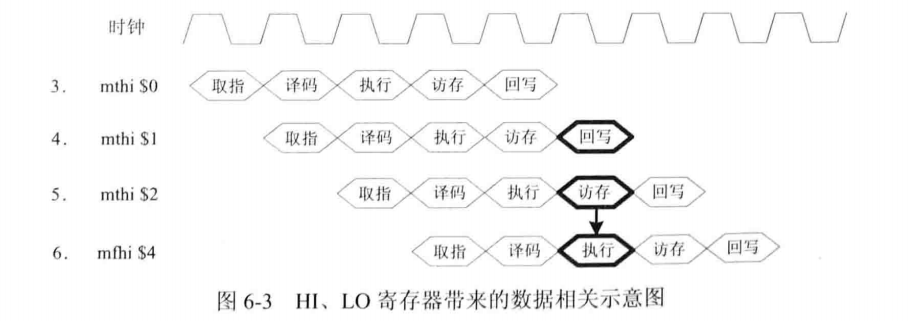
我们在回写阶段增加HILO寄存器模块，如图：



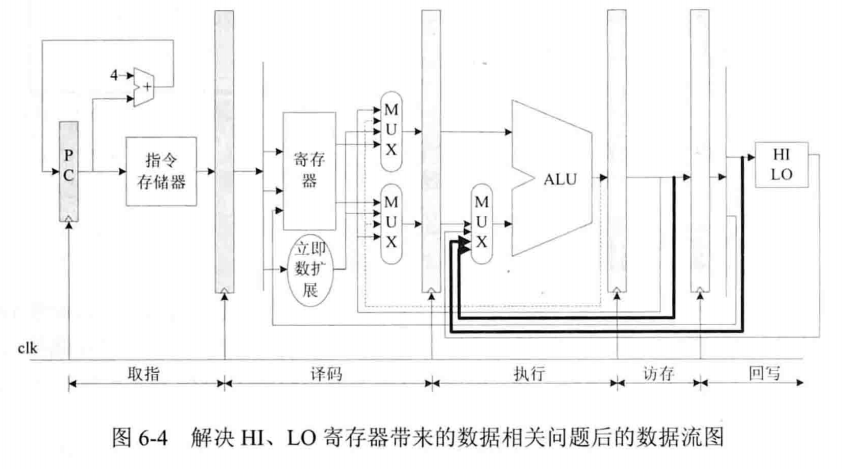
# 出现新的数据相关

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：

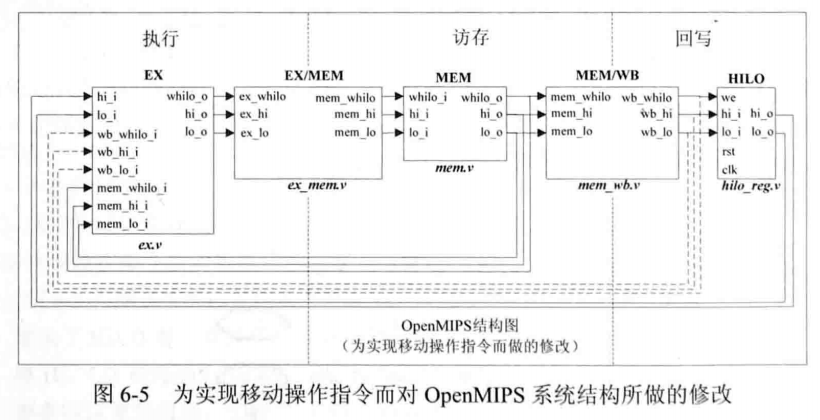




我们发现，在第六步的执行阶段读取到HILO寄存器的值时，第四步才刚刚回写，第五步还没有回写，所以获取到了是第四步回写进HILO的值，而正确的应该是获取到第五步中回写进HILO的值。这便产生了数据相关，解决办法还是数据前推。



因此我们进行系统结构的修改，如图：



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。

# HI、LO寄存器的实现

接口描述如表：



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

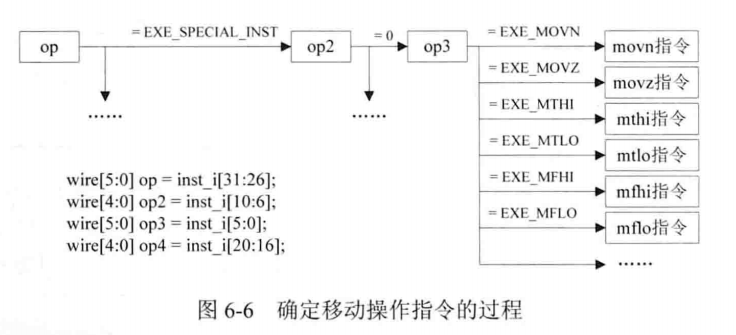
end

endmodule

很简单，没什么说的。

# 修改译码阶段的ID模块

相关指令的功能码对照：



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。

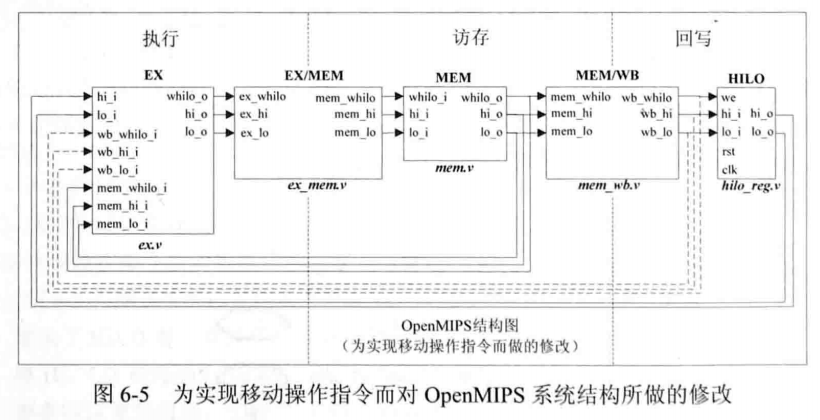
另外，movz指令的译码过程需要读取rs和rt寄存器的值，所以reg1\_read\_o和reg2\_read\_o都是1，默认通过RegFile模块读端口1读取的寄存器地址reg1\_addr\_o的值是指令的21~25bit，正是movz指令中的rs，默认通过RegFile模块及短期地址reg2\_addr\_o的值是指令的16~20bit，正是movz指令中的rt。所以如果reg2\_o为0，则wreg\_o为WriteEnable。

# 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：



这些接口便实现了数据前推，避免了数据相关问题。



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;

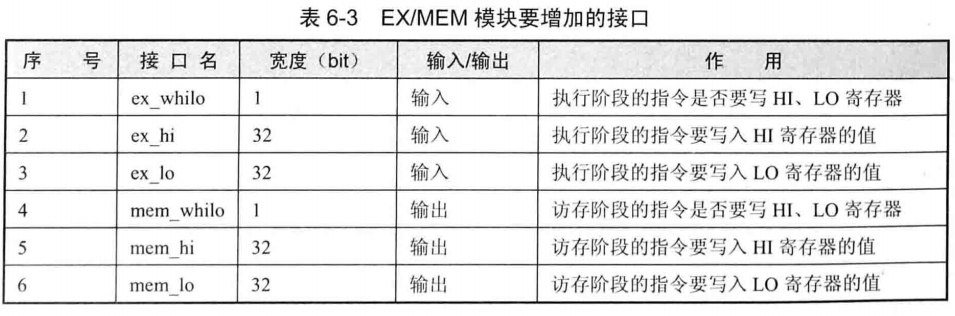
lo\_o <= `ZeroWord;

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：



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

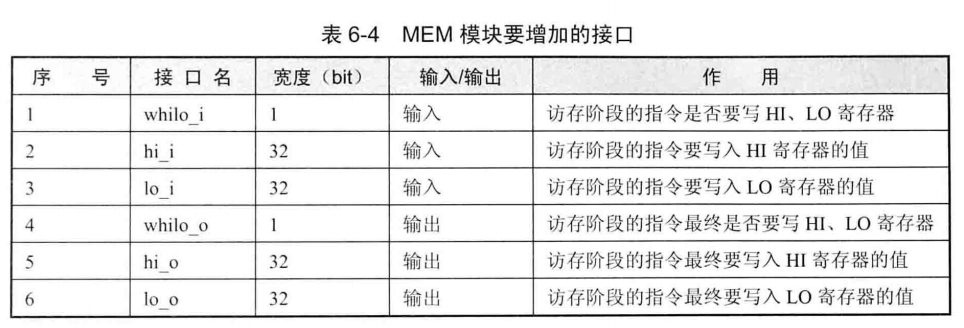
end

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：



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

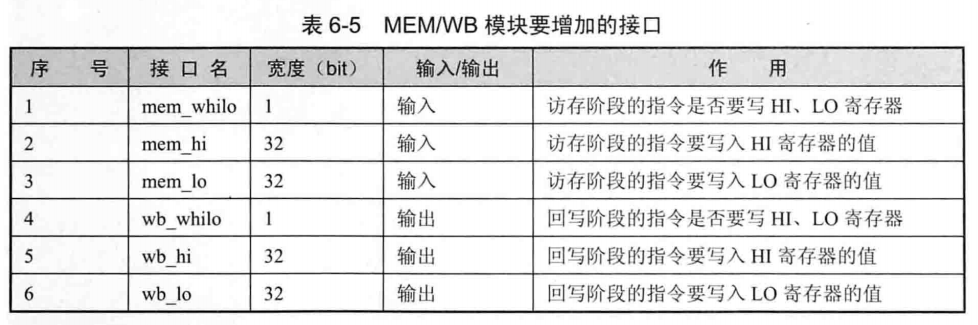
end

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：



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

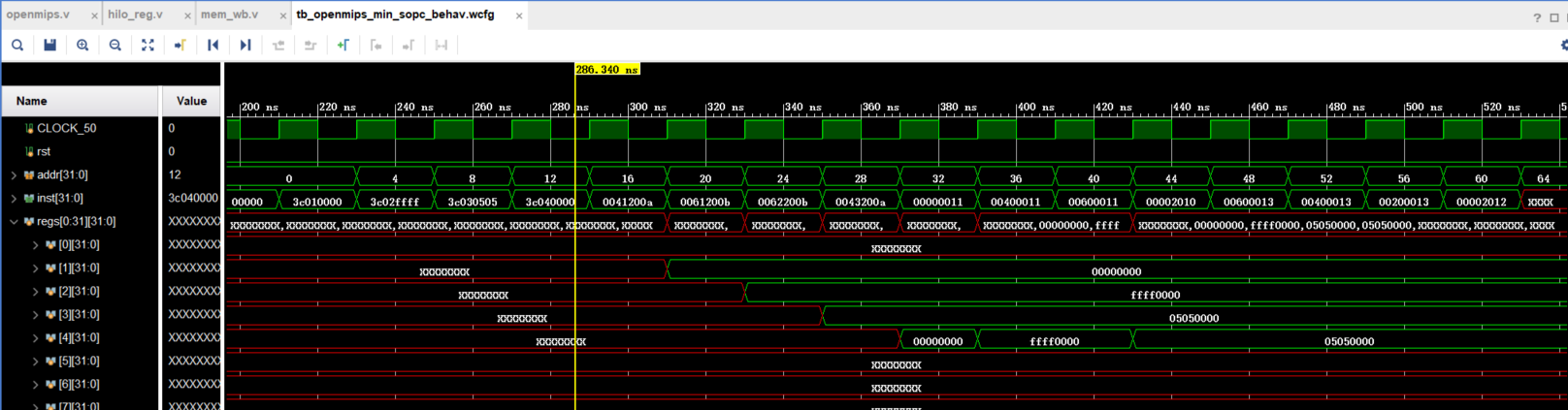
mtlo $3 # lo = 0x05050000

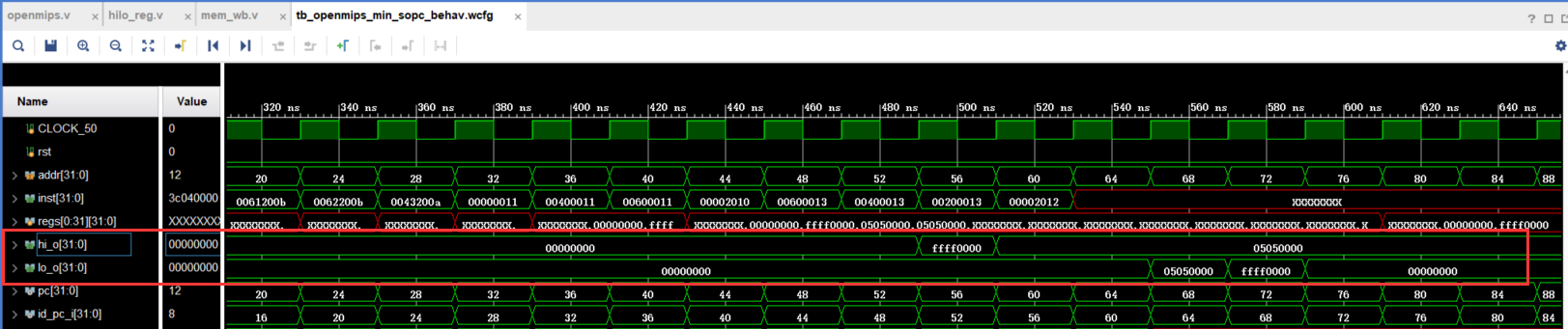
mtlo $2 # lo = 0xffff0000

mtlo $1 # lo = 0x00000000

mflo $4 # $4 = 0x00000000

仿真结果如下：

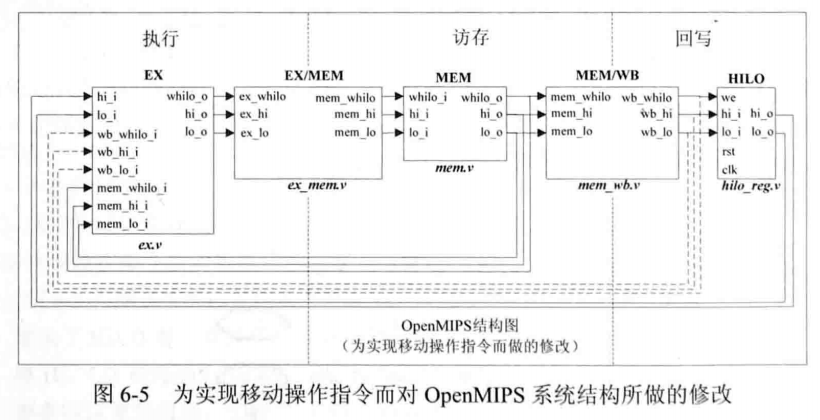




# 十一、总结

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：



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。