**加载存储指令的实现**

本章分两步，除了ll、sc指令外的一般加载存储指令，其次就是比较特殊的ll、sc加载存储指令。

# 加载存储指令的说明

## 加载指令 lb、lbu、lh、lhu、lw 说明



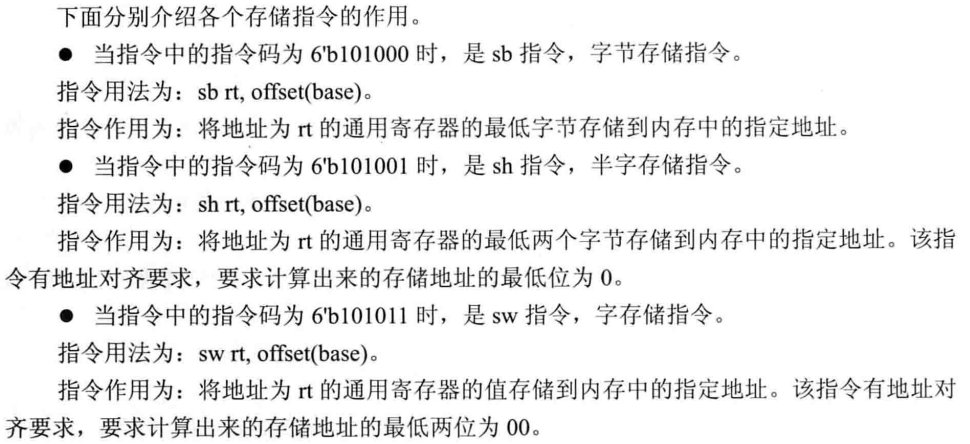
我们得到：

 其中 offset 是段地址，base 是偏移地址。

指令使用说明：

## 存储指令 sb、sh、sw 说明



我们得到：

至于为什么有地址对齐要求，最后一位为0，说明地址必须能被2整除，最后两位为0说明地址被4整除。因为地址按照字节编址和寻址，因此一个32位的MIPS的字要占四个内存单位，半字要占两个内存单位。

## 使用示例

## 加载指令lwl、lwr 说明

指令用法：**lwl rt, offset(base)**

该指令稍微复杂一些，当指令码为6’b100010 时，是lwl指令，非对齐加载指令，向左加载。指令用法依然是将段地址和偏移地址表示的内存的值送到寄存器rt所表示的内存。

该指令的作用为：从内存中指定的加载地址处，加载一个字的最高有效部分。lwl指令对加载地址没有要求，也就是说允许地址最后一位不为0，也就是允许非对齐加载，这跟前面的lh、lhu、lw等指令不太一样。因此我们可以想到，该指令对于大端地址方式和小段方式的结果是不一样的，但是因为MIPS是大端存放的地址，因此这里默认是大端存放。

现在假设计算出来的加载地址为 loadaddr，loadaddr的最低两位的值为n，将loadaddr最低两位设为0后的值称为loadaddr\_align，即：

**加载地址 loadaddr = signed\_extended(offset) + GPR[base].**

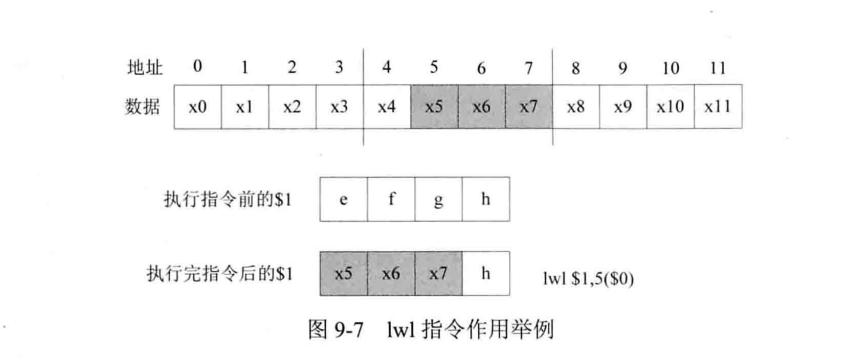
**n = loadaddr[1:0]**

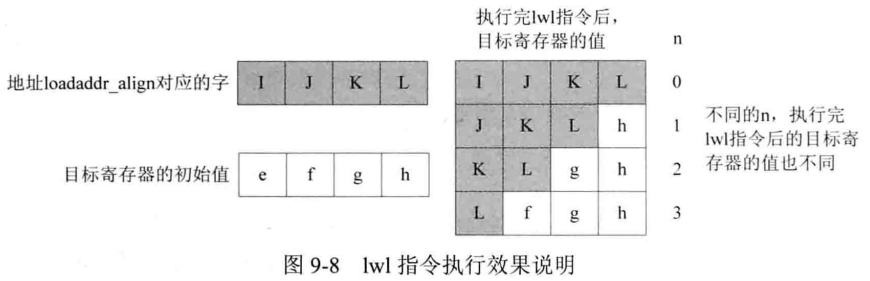
**Loadaddr\_align = loadaddr – n**

下面举个例子：

加入计算出来的加载地址为5，lwl指令要从地址5加载数据，那么loadaddr就等于5，n等于1，loadaddr\_align等于4.

lwl指令的作用是从地址为loadaddr\_align处加载一个字，也就是4个字节，然后将这个字的最低**4-n**个字节保存到地址为rt的通用寄存器的高位，并且保持低位不变，如下图：

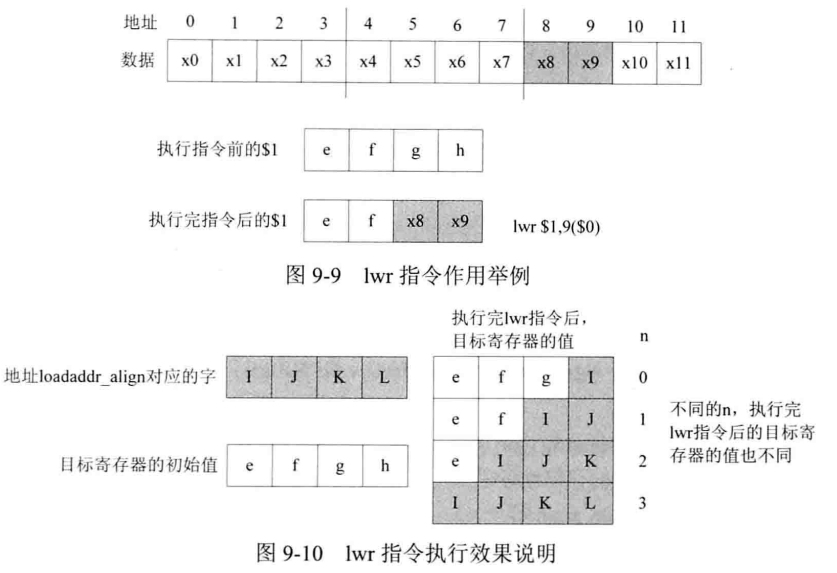


 lwl指令执行效果说明：

同理，lwr指令加载一个字的最低有效部分，从地址为loadaddr\_align处加载一个字，也就是4个字节，然后将这个字的最高**n+1**个字节保存到地址为rt的通用寄存器的低位，并且保持高位不变。

指令用法：**lwr rt, offset(base)**

如图：



我们通过lwl和lwr指令配合可以实现从个非对齐地址加载一个字，而且只需要使用2条指令就可以实现。比如若要从地址7处加载一个字，那么可以使用一下代码来实现，共5条指令：

**lw $1, 4($0) #取得地址0x4处的字，保存在$1中**

**lw $2, 8($0) #取得地址为0x8中的字，保存在$2中**

**sll $1, $1, 24 #$1左移24位**

**slr $2, $2, 8 #$2右移8位**

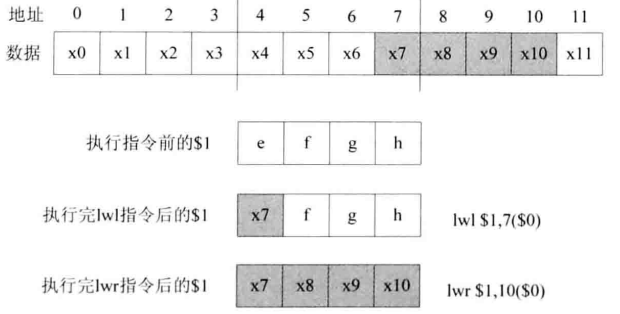
**or $1, $1, $2 #$1与$2进行逻辑“或”运算，得到最终结果**

但是有了lwl、lwr指令后，只需要两条指令即可：

**lwl $1, 7($0)**

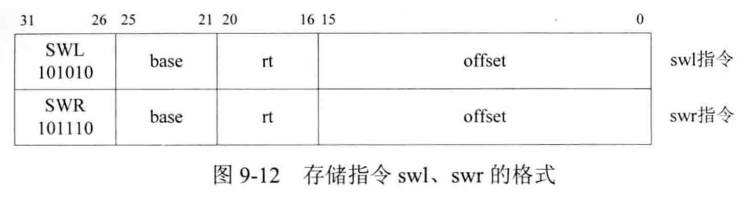
**lwr $1, 10($0)**

如图所示：



## 存储指令swl、swr说明

格式如下图所示：



该指令和lwl、lwr相反，是load存储指令，作用是将地址为et的通用寄存器的高位部分存储到内存中指定的地址处，存储地址最后两位确定了要存储rt通用寄存器的哪几个字节。swl指令对存储地址没有对齐要求，这点和lwl、lwr相似。

假设计算出来的存储地址为storeaddr，storeaddr最低两位的值为n，storeaddr最低两位设为0后的值称为storeaddr\_align，如下：

**存储地址 storeaddr = signed\_extended(offset) + GPR[base]**

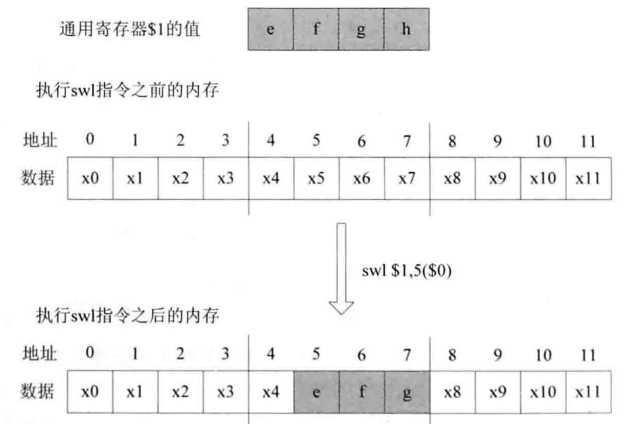
**n = storeaddr[1:0]**

**storeaddr\_aglin = stroeaddr – n**

举个例子：

计算出来的存储地址为5，swl指令要向地址5存储数据，那么storeaddr就等于5，n等于1，storeaddr\_align等于4.

swl指令的作用是将地址为rt的通用寄存器的最高**4-n**个字节存储到storeaddr处，与lwl类似，如下图：



同理，swr指令，非对齐存储指令，向右存储。

指令用法：**swl rt, offset(base)**

指令执行效果说明如下图所示：



swr指令与lwr相似，故略。

利用swl、swr指令配合可以实现一个非对齐地址存储一个字，也只需要用两条指令，例如在地址为7的地方存储一个字，常规方法需要5条指令：

**sll $2, $1, 24 #将$1的最高字节存储到$2**

**sb $2, 7($0) #存储最高字节到地址为7的内存处**

**sll $2, $1, 8 #将$1的第2、1字节保存到$2中**

**sh $2, 8($0) #存储第2、1字节到地址为8、9的内存处**

**sb $1, 10($0) #存储第0字节到地址为10的内存处**

但是利用swl、swr只需要2两条指令：

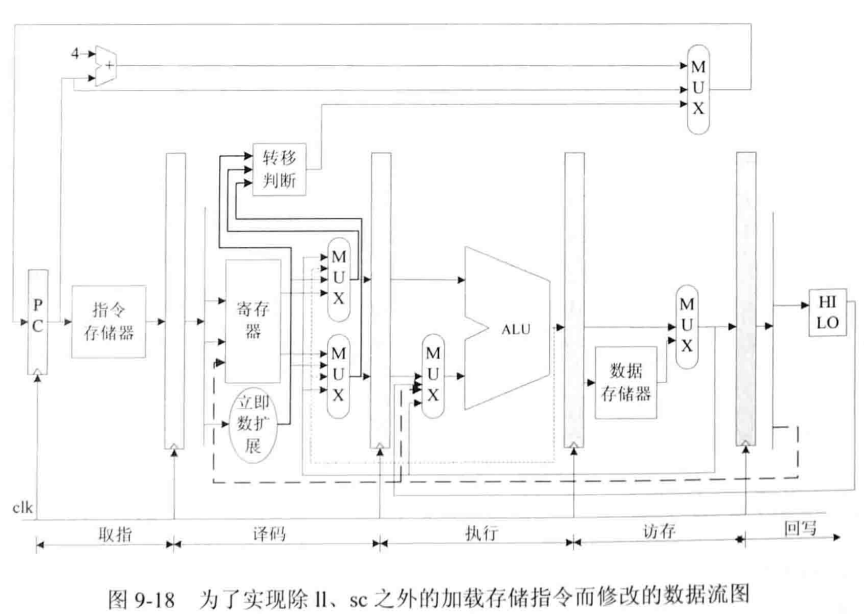
**swl $1, 7($0)**

**swr $1, 10($0)**

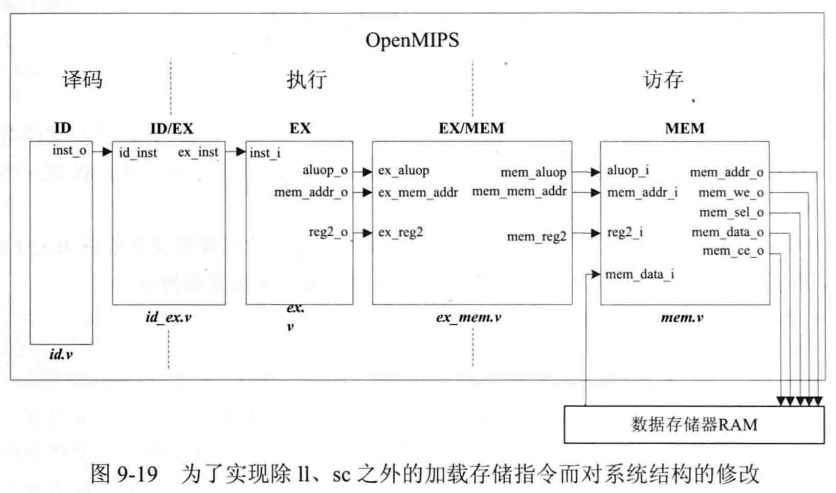
简单来说，这类指令，包括lwl、lwr和swl、swr可以实现任意非对齐地址的存储和加载，只要将指令（lwl和swl）设置为想要加载（或储存）的地址的左边界，将指令（lwr和swr）设置为想要加载（或储存）的地址的右边界即可。

# 加载存储指令的实现思路

由于加载和存储指令都是在译码阶段进行译码得到运算类型以及要写入的目的寄存器等信息，最终在访存阶段根据这些信息设置对数据存储器RAM的访问信号，将数据写入RAM，因此在访存阶段增加对数据存储器RAM的访问，同时，由于要写入的目的寄存器的数据可能是执行阶段的结果，也可能是在访存阶段从数据存储器RAM加载得到的数据，所以在访存阶段增加了一个多路选择器：



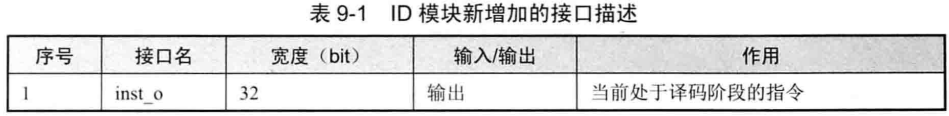
我们也将系统结构进行修改：



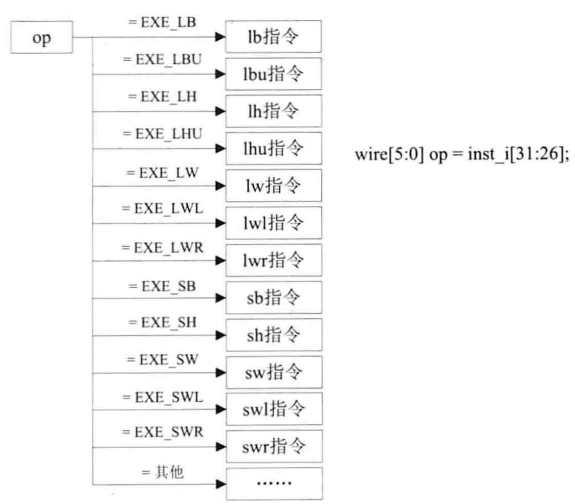
# 修改代码具体实现

## 3.1 修改ID模块

增加接口如下表：



我们根据不同指令的不同的指令码来加以区分，如下图所示：



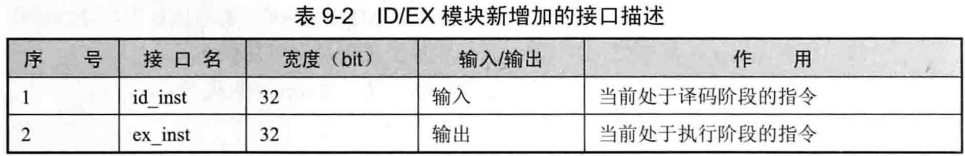
下面对ID模块修改的代码进行几点说明：

1. 加载指令需要将加载的结果写入目的寄存器，因此寄存器写使能信号要打开，为WriteEnable；存储指令不需要写入目的寄存器，因此寄存器写使能信号为WriteDisable。
2. 对于包括lb、lbu、lh、lhu、lw等指令，只需要读取一个寄存器；
3. 要写入的目的寄存器的地址为inst\_i[20:16]，因此输出数据wd\_o为inst\_i[20:16]；
4. 对于lwl、lwr、swl、swr指令，需要读取两个寄存器，一个是base表示的寄存器，对于另外一个，由于lwl等指令只是修改目的寄存器的部分，因此还需要读出目的寄存器的值。reg1\_o表示地址为base的寄存器的值，reg2\_o表示地址为rt的寄存器的值；存储指令也类似，需要读取两个寄存器的值分别是rt和base.

代码略。

## 3.2 修改ID/EX模块

新增加的接口描述如下表：



该模块将id模块的指令信号传递到ex模块，很简单。

## 3.3 修改EX模块

新增加的接口描述如下表：



看如下代码：

assign aluop\_o = aluop\_i;

assign mem\_addr\_o = reg1\_i + {{16{inst\_i[15]}}, inst\_i[15:0]};

assign reg2\_o = reg2\_i;

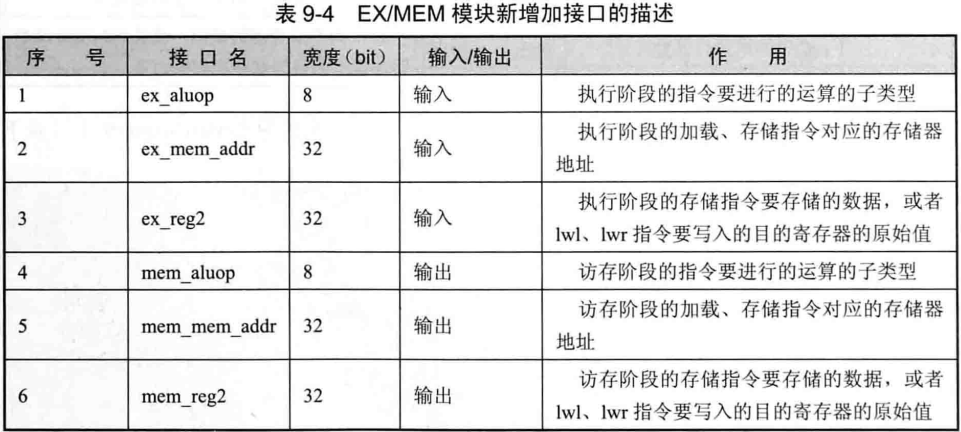
第一行表示将运算类型传递到访存阶段，确定加载存储类型；

第二行计算出存储器地址，也就是offset[base]的值，其中reg1\_o是base的值，inst\_i[15:0]是offset的值，然后将inst\_i做符号位扩展成32位；

第三行中的reg2\_i是存储指令要存储的数据，或者是lwl、lwr指令要加载到目的寄存器的原始值。

## 3.4 修改EX/MEM模块

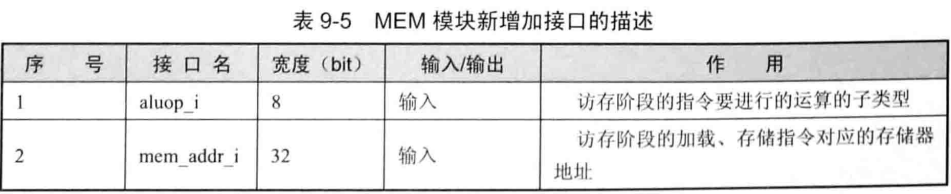
增加接口描述如下表：



该模块主要将EX模块的输入传递到访存阶段，也很简单。

## 3.5 修改MEM模块

增加的接口描述如下表：





这些接口大部分和数据存储器相连，具体介绍一下mem\_sel\_o接口：

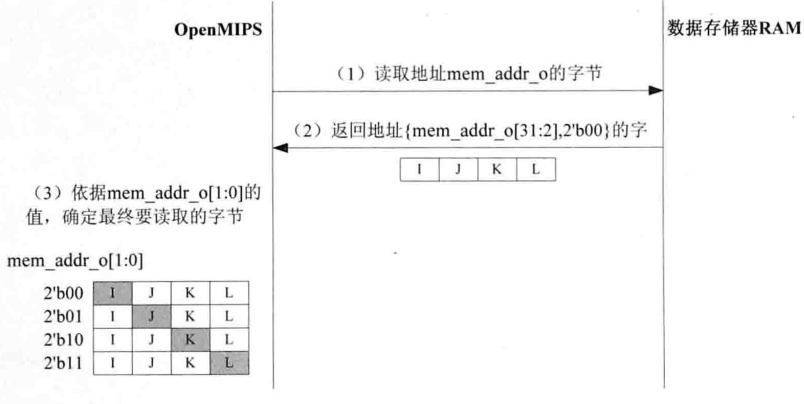
首先对于加载操作，MIPS32指令集架构可以加载字、半字、字节，但是数据总线宽度是32位，只有4个字节，因此在MIPS架构中1个字就是4字节。如果是lb、lh等加载指令，我们需要知道是加载了一个字中的哪个字节，mem\_sel\_o接口就是指出一个字中的有效加载数据是第几字节。举个例子，如果mem\_sel\_o为4’b0100，表示处理某个字的次高位，也就是16-23 bit.

对于存储操作，若使用存储指令sh向地址为0x2处存储0x8281，设置mem\_data\_o为0xffff8281、设置mem\_sel\_o为4’b0011，这样外部存储器就会将最低两个字节进行存储。

一开始我有个疑问，要是说lwl、lwr、swl、swr等地址不对齐的指令用mem\_sel\_o进行字节定位还可以理解，为什么连lb这种本身就是对字节进行加载的指令也要用mem\_sel\_o进行定位操作，难道仅仅是为了统一吗？

一个原因可能是为了统一规范，另一个原因我们来慢慢分析：

先用lb指令来举例，我们要访问数据存储器，因此mem\_ce\_o为ChipEnable，加载操作，设置mem\_we\_o为WriteDisable，给出要访问的数据存储器的地址mem\_addr\_o，其值就是执行阶段算出来的地址mem\_addr\_i，根据地址的最后两位来确定mem\_sel\_o的值。见下图：当地址为的最后两位为00的时候，即地址为4的倍数，因此设置mem\_sel\_o为4’b1000，同理，当地址最后两位为01的时候，设置mem\_sel\_o为4’b0100…

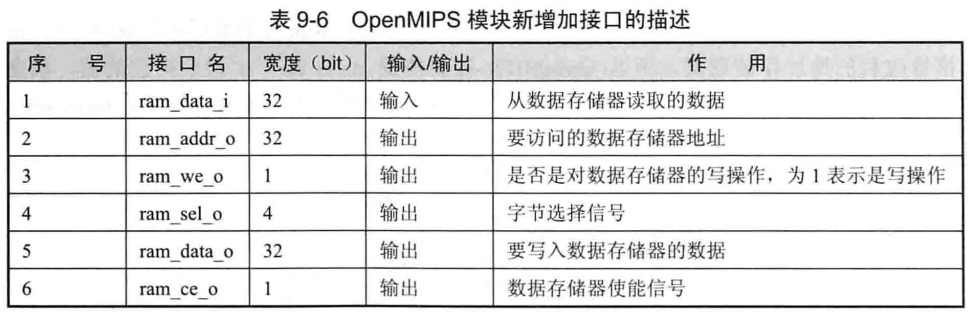


对于lwl、lwr等指令，我们要将mem\_addr\_i的最后两位置0，是因为lwl指令要从RAM中读出一个字，所以要将地址对齐，同时设置mem\_sel\_o为4’b1111。

对于lb等指令，我们可以直接设置mem\_sel\_o为4’b0001，告诉数据存储器读取 0-7 bit，这样似乎更简单，那为什么不这样设呢？书上给了我们答案：这里确定mem\_sel\_o值的过程参考了Wishbone总线的相关规范，为的是在后期给OpenMIPS添加Wishbone总线接口的时候容易一些。OpenMIPS总是根据mem\_addr\_o的最后两位确定要读取的字节。

## 3.6 修改OpenMIPS顶层模块

增加的接口描述如下表：

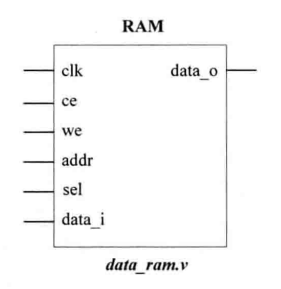


处理器接口图：

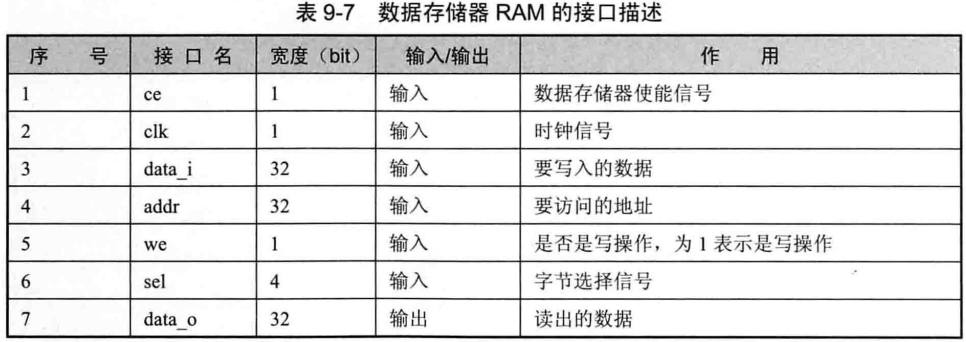


## 3.7 添加数据存储器RAM

模块接口图：



接口描述如下表：



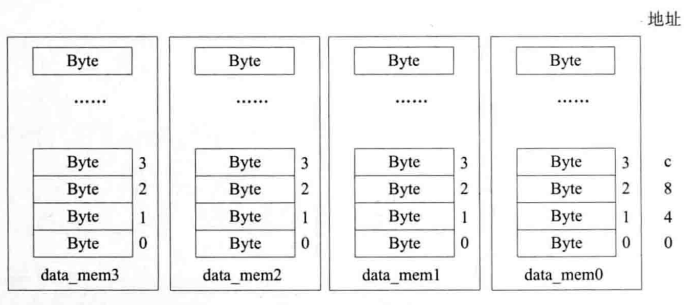
数据存储器中，为了方便对数据进行字节寻址，在设计的时候用4个8位的存储器代替一个32位的存储器，如以下代码：

**reg[`ByteWidth] data\_mem0[0:`DataMemNum - 1];**

**reg[`ByteWidth] data\_mem1[0:`DataMemNum - 1];**

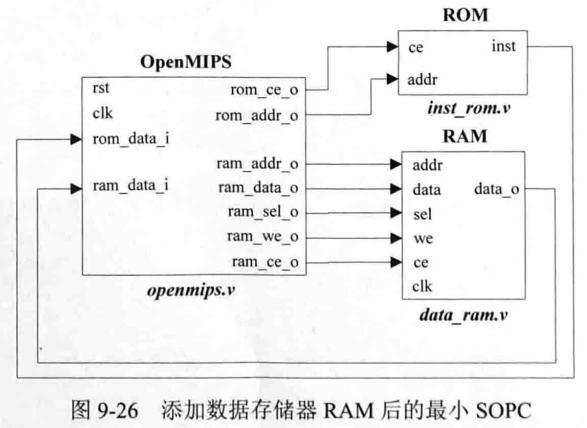
**reg[`ByteWidth] data\_mem2[0:`DataMemNum - 1];**

**reg[`ByteWidth] data\_mem3[0:`DataMemNum - 1];**

就是分为了4个8位寄存器，读操作时，从4个8位寄存器中各读出一个字节，组合成为一个32位的数，比如读地址为n的字，就是读取每个寄存器地址为n/4的字节，组合起来成为地址n处的字。写操作时，根据sel的值来确定写入哪个8位寄存器，如下图：

## 3.8 修改最小SOPC

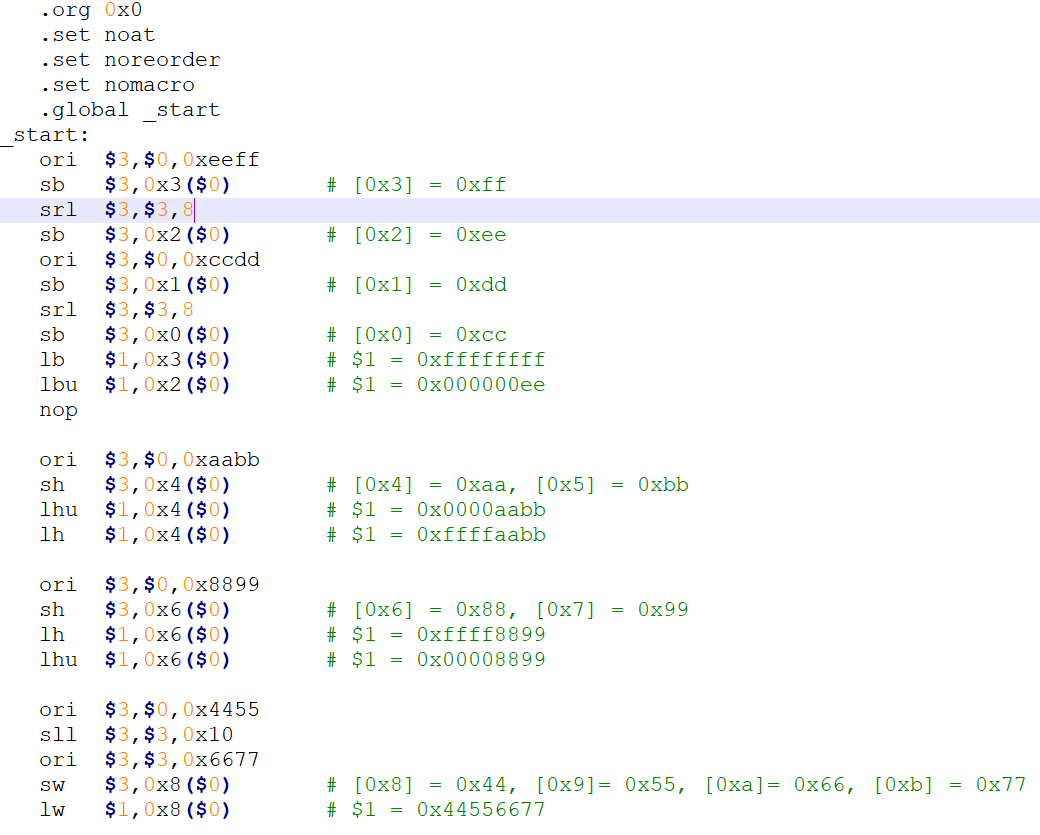
添加RAM后的最小SOPC：

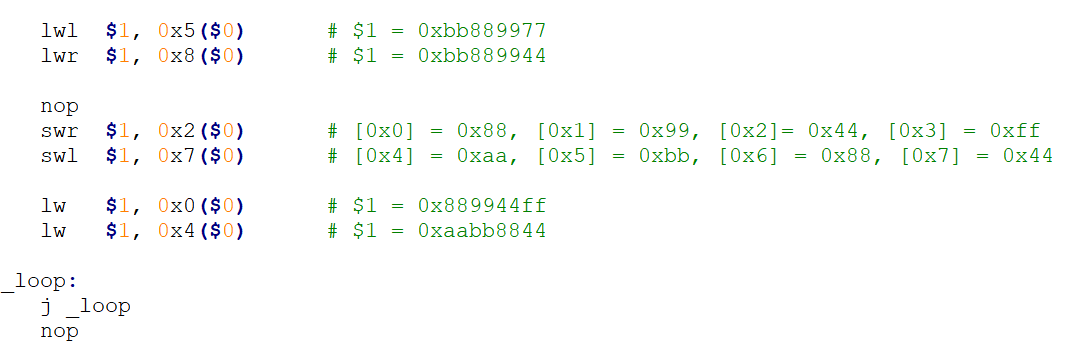


# 测试及结果分析

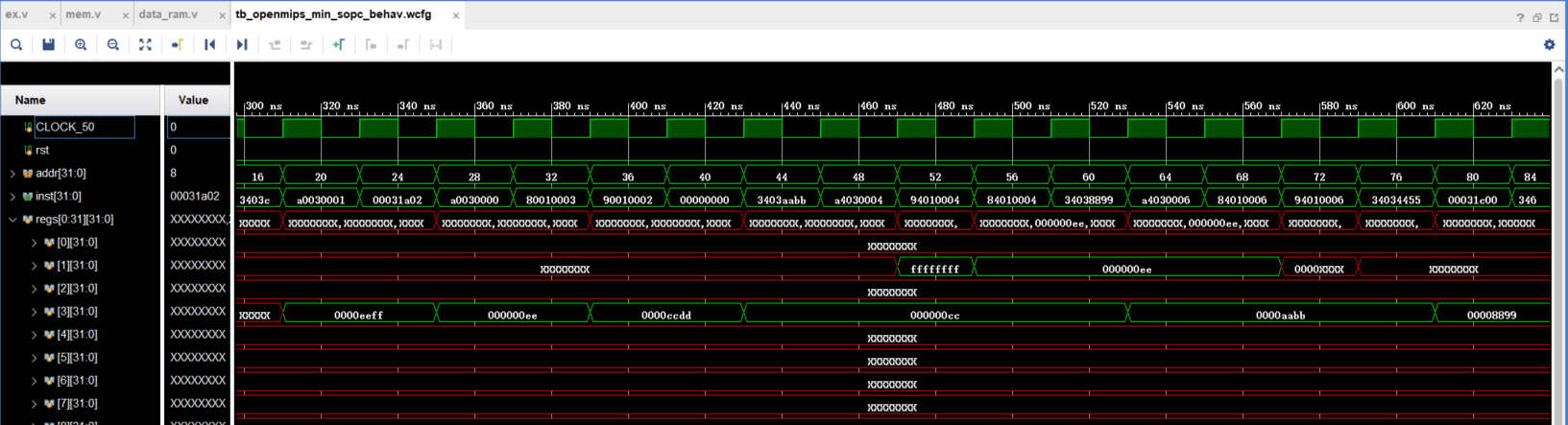
## 4.1 测试

使用以下测试程序：





## 4.2 仿真结果





查看寄存器的值，结果正确。

# 链接加载指令ll、条件存储指令sc说明

我们用链接加载指令ll、条件存储指令sc来实现信号量机制。

信号量机制是操作系统中为了避免不同进程同时访问一个临界资源而导致资源混乱而产生的一个方法，基本的信号量操作是创建一个信号量semaphore，然后对信号量进行PV操作：

**wait(semaphore);**

**atom operation**

**signal(semaphore);**

ll指令和一般的加载指令一样，从内存中加载一个字，但是ll指令还会降处理器内部的一个链接状态位LLbit置为1，表明发生了一个链接加载操作，并将链接加载的地址保存到一个特殊寄存器LLAddr中（这个寄存器在多处理器中有作用，OpenMIPS是单处理器，故没有该寄存器）。

ll指令执行完毕后，会进行一定的操作，然后执行sc指令，这可以认为是一个RMW序列。有如下两种情况干扰这个RMW序列，受到干扰后，处理器会设置链接状态位LLbit为0.

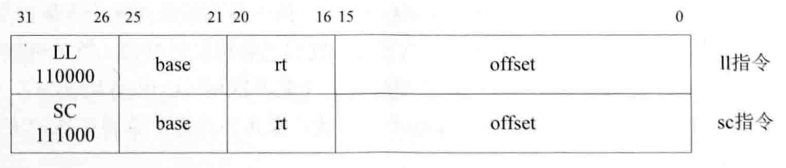
·在ll、sc指令之间产生异常，从而进入异常处理例程，或者发生线程切换，导致RMW序列受到干扰。

·多处理器的系统中，另一个CPU改写了RMW序列要操作的内存空间。

当然对于这个OpenMIPS处理器，只有第一种情况。

执行sc指令时，会对从ll指令开始的RMW序列开始检查，判断是否受到干扰，实际就是判断LLbit是否为1，如果没有受到任何干扰，LLbit保持为1，那么操作是原子的，sc指令会对ll指令加载数据的地址进行写回操作，并设置一个通用寄存器的值为1，表示成功，反之不进行写回操作，并设置一个通用寄存器的值为0，表示失败。

ll、sc指令格式如下表所示：

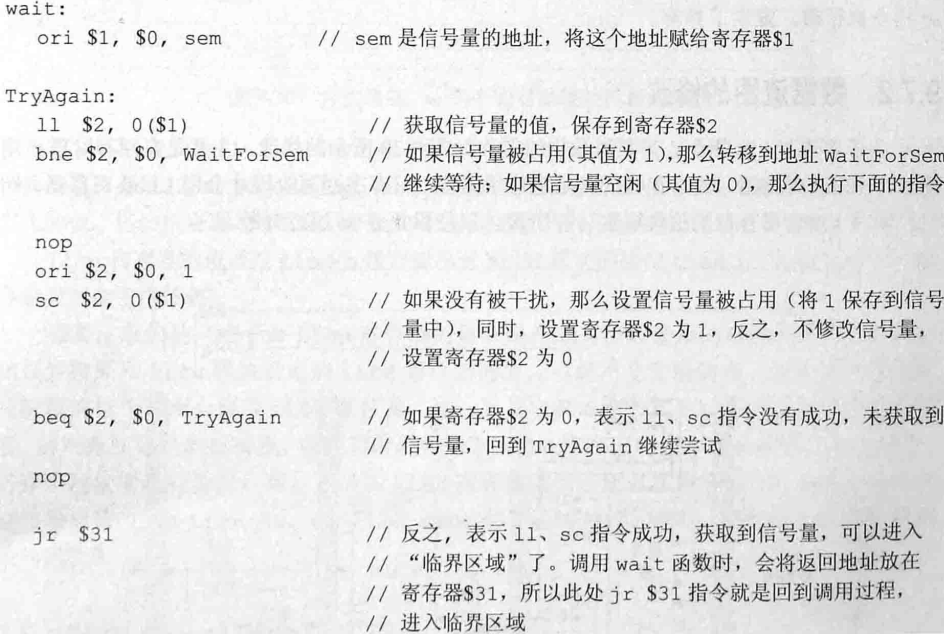


ll指令用法为: ll rt, offset(base)

ll指令作用为: 从内存中指定的加载地址处，读取一个字节，然后符号拓展至32位，保存到地址为rt 的通用寄存器中，加载地址的计算方式与其他加载指令一致，同时设置链接状态位LLbit为1.

sc指令作用为: 如果RMW序列没有受到干扰，也就是LLbit为1，那么将地址为rt的通用寄存器的值保存到内存中指定的存储地址处，同时设置地址为rt的通用寄存器的值为1，设置LLbit为0.如果RMW序列受到了干扰，也就是LLbit为0，呢嘛不修改内存，同时设置地址为rt 的通用寄存器的值为0.

下面举例说明ll、sc的作用，并实现wait操作：



## 5.1 ll、sc指令实现思路

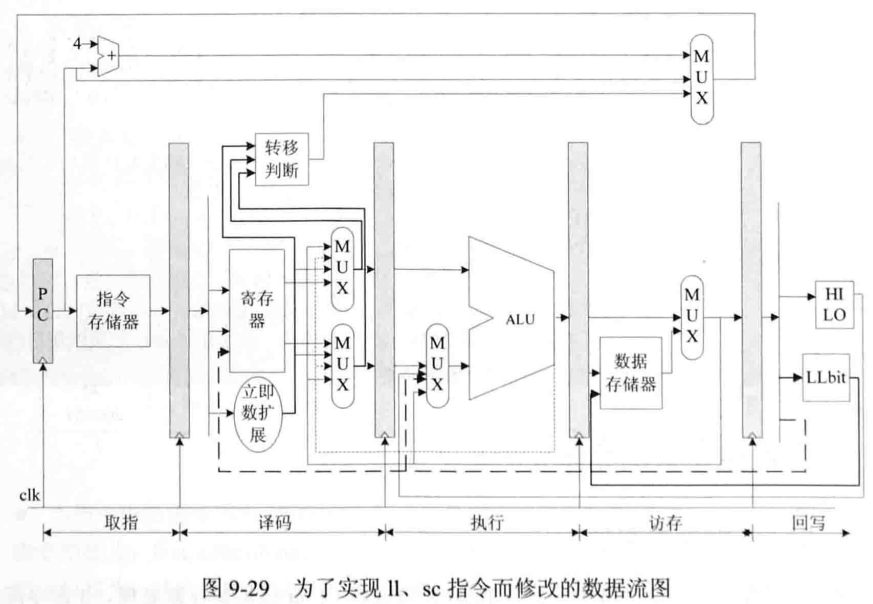
这两条指令都要访问链接状态位LLbit，故可以增加一个寄存器，对LLbit寄存器的写操作也放在回写阶段进行。

sc指令在访存阶段获得LLbit寄存器的值，如果值为1，那么完成存储操作，同时设置对LLbit的写操作，写入的值为0，还要设置对通用寄存器rt的写操作，写入的值为1，这些写操作都会通过MEM/WB模块传递到回写阶段，最终实现对寄存器LLbit和通用寄存器rt的写操作。如果LLbit寄存器的值为0，则不进行存储操作。

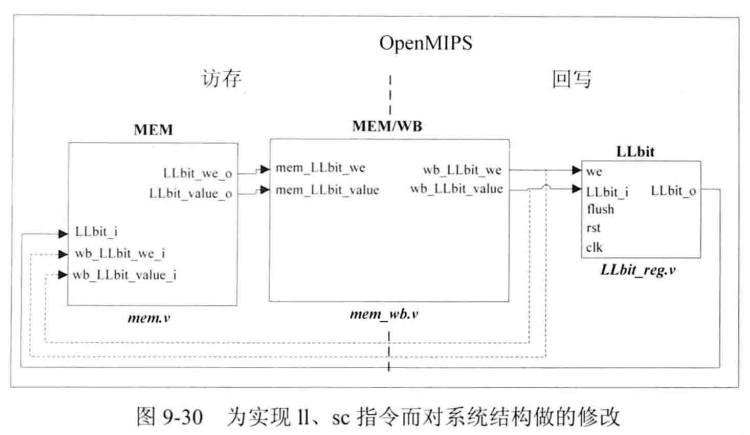
导致寄存器LLbit的值为0的情况有：（1）sc指令之前没有执行ll指令；（2）ll指令执行后、sc指令执行前发生了异常。

## 5.2 数据流图的修改

主要在回写阶段新增了LLbit寄存器



## 5.3 系统结构的修改



值得注意的是，因为sc指令是在访存阶段判断LLbit寄存器的值，而ll指令是在回写阶段才写入，这样会产生数据相关问题，我们用数据前推方法来解决这一问题。因此MEM/WB模块的输出信号wb\_LLbit\_we、we\_LLbit\_value也要送到MEM模块。

# 修改代码以实现ll、sc指令

## 6.1 LLbit寄存器的实现

接口描述如下：



module LLbit\_reg(

input wire clk,

input wire rst,

//Whether exception happened

input wire flush,

//write

input wire LLbit\_i,

input wire we,

//value of LLbit reg

output reg LLbit\_o

);

always @ (posedge clk) begin

if (rst == `RstEnable) begin

LLbit\_o <= 1'b0;

end else if (flush == 1'b1) begin

LLbit\_o <= 1'b0;

end else if (we == `WriteEnable) begin

LLbit\_o <= LLbit\_i;

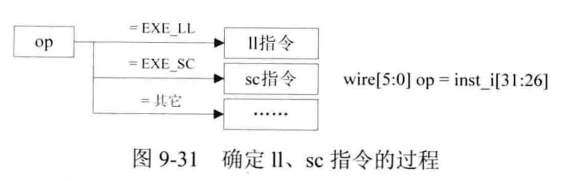
end

end

endmodule

代码也很简单，当有异常发生时，会使得LLbit寄存器的值为0，设置一个flush接口，当flush为1时，表示有异常发生。

## 6.2 修改ID模块



## 6.3 修改MEM模块

接口描述如表所示：



我们在代码中新增加了一个reg型变量LLbit，用来获取LLbit寄存器的最新值，如果回写阶段的指令要写LLbit，那么回写阶段要写入的值就是LLbit寄存器的最新值，反之，LLbit模块给出的值LLbit\_i是最新值，这就避免了数据相关。

右边的代码中，wdata\_o是写入寄存器中的值，因为要置地址为rt的通用寄存器的值为1，因此要设置wdata\_o的值为1.

`EXE\_SC\_OP: begin

if (LLbit == 1'b1) begin

LLbit\_we\_o <= 1'b1;

LLbit\_value\_o <= 1'b0;

wdata\_o <= 32'b1;

mem\_addr\_o <= mem\_addr\_i;

mem\_we <= `WriteEnable;

mem\_data\_o <= reg2\_i;

mem\_sel\_o <= 4'b1111;

mem\_ce\_o <= `ChipEnable;

end else begin

wdata\_o <= 32'b0;

end

end

因为要访问数据寄存器，所以设置mem\_ce\_o为ChipEnable.

要设置LLbit寄存器为0，因此要设置LLbit\_we\_o为1，表示要写入LLbit寄存器，同时设置LLbit\_value\_o的值为0.

反之，如果LLbit的值为0，表示之前没有执行过ll指令，或者在ll指令指向后、sc指令执行前有异常发生，此时sc指令的访存信息如下：

·不修改数据寄存器，因此mem\_we\_o保持默认值WriteDisable，mem\_ce\_o保持默认值ChipDisable，不修改LLbit寄存器的值，因此Llbit\_we\_o保持默认值0，要设置地址为rt的通用寄存器为-，所以设置wdta\_o为0.

## 6.4 修改MEM/WB模块

接口描述如表所示：

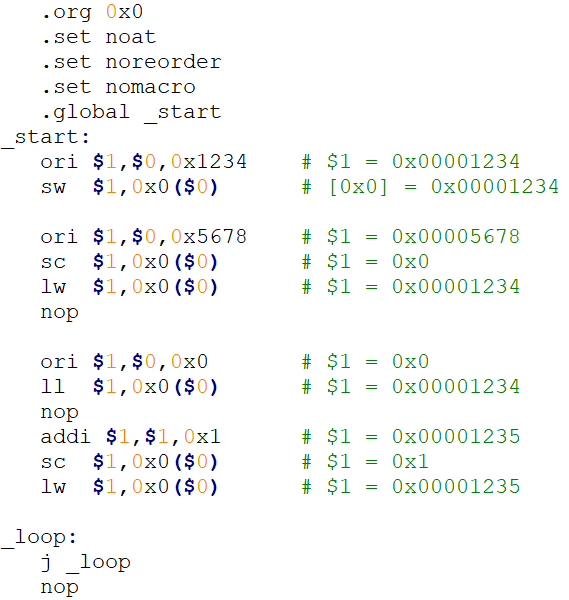


## 6.5 修改OpenMIPS顶层模块

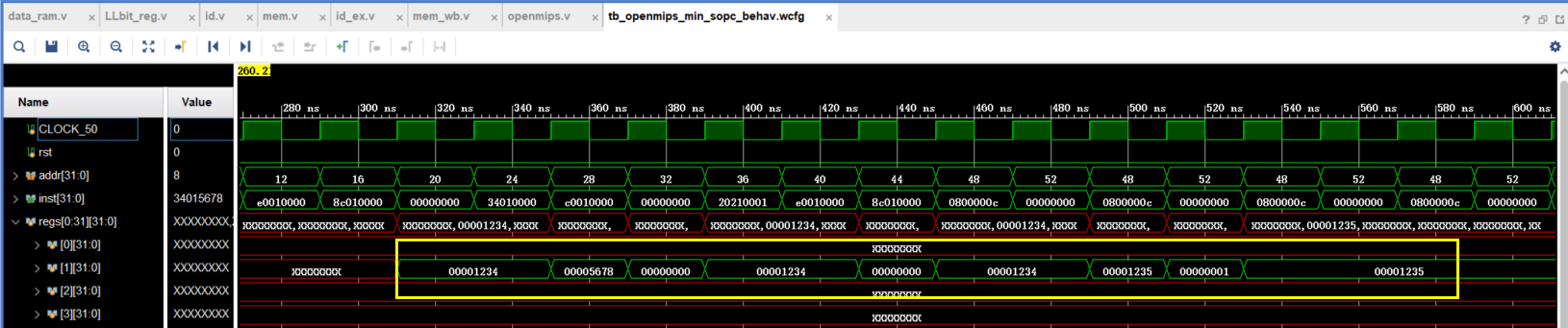
略

# 测试ll、sc指令

使用下面测试代码：



仿真结果为：



结果正确。

# Load相关问题

## 8.1 问题介绍

观察下面这段程序：

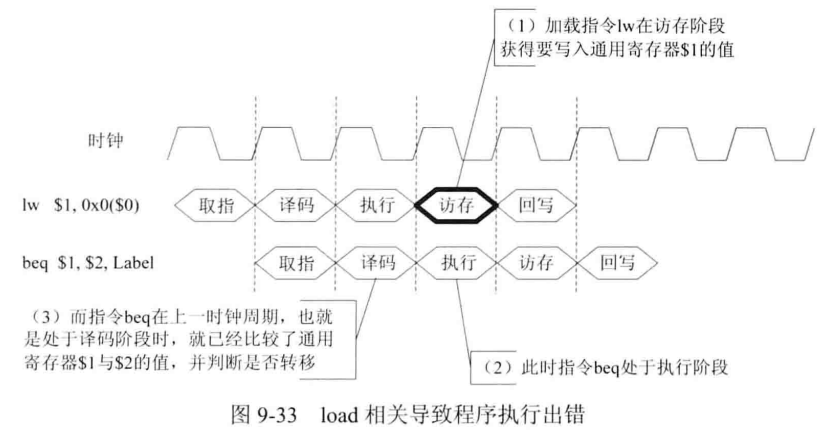
**...**

**lw $q, 0x0($0) //从数据存储器的地址0x0处加载字，保存到通用寄存器$1**

**beq $1, $2, Lable //比较通用寄存器$1与$2，如果相等，转移到标号Lable处**

**...**

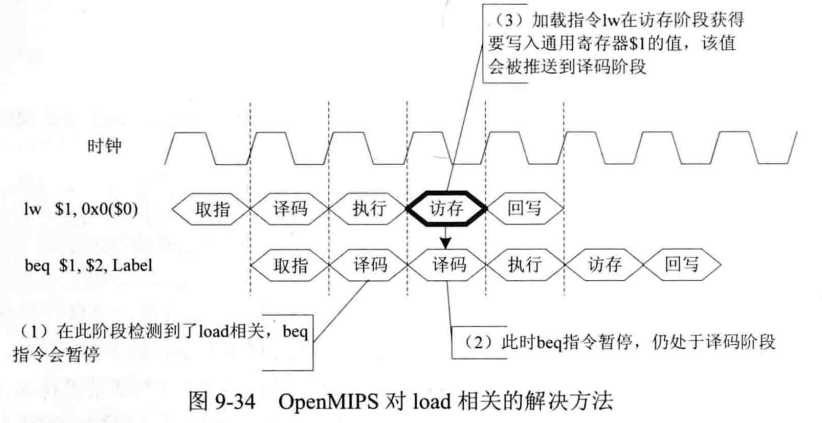
加载指令lw会在访存阶段从数据存储器读取数据，也就是在访存阶段才能获得要写入的通用寄存器$1的值，这个值是$1的最新值，此时紧接着的转移指令beq处于执行阶段，而beq在上一周期译码阶段时，就已经对寄存器$1和$2的值进行了比较，并判断是否转移，显然这两个寄存器值的比较先于lw指令，因此程序运行是紊乱的，如下图所示：

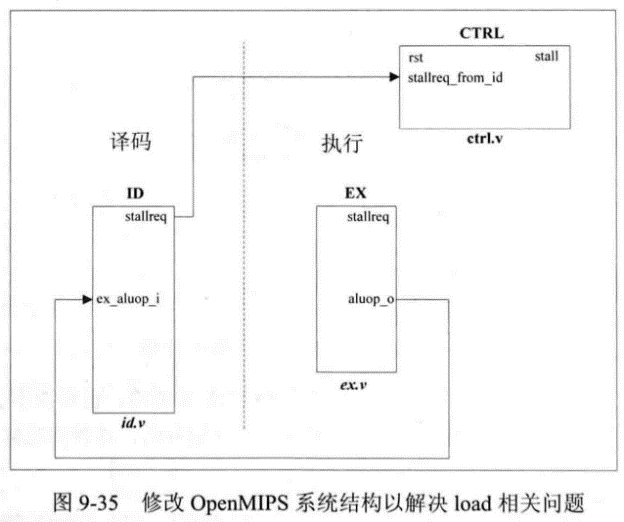


即使通过数据前推的方法，将访存阶段加载得到的数据前推，也解决不了问题，因为数据加载时，beq指令已经处于执行阶段了，已经进行了比较判断，这种情况叫做load相关。

## 8.2 解决办法

在译码阶段检查当前指令与上一条指令是否存在load相关，如果存在，就让流水线的译码、取指阶段暂停，而执行、访存、回写阶段继续，相当于插入一条空指令，处于执行阶段的加载指令继续运行，当运行到访存阶段时，将加载得到的数据前推到译码阶段，然后流水线继续运行，如下图所示：



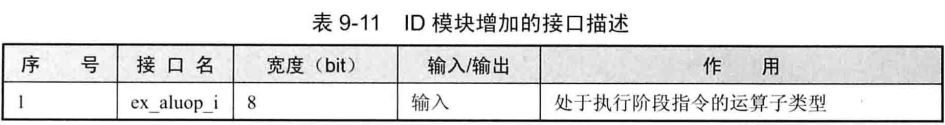
 我们已经实现了将访存的数据前推到译码阶段，下面我们来实现判断load相关的问题。

我们对系统结构做如下修改：

将处于执行阶段的指令的运算子类型aluop\_o、要写的目的寄存器地址wd\_o等信息传递到译码阶段的ID模块，后者根据运算子类型判断是否存在load相关，若存在，通过stallreq接口通知CTRL模块请求流水线暂停。

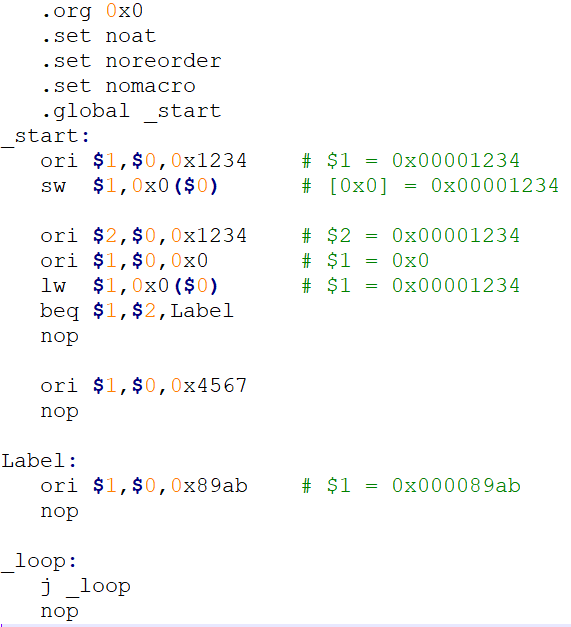
## 8.3 修改OpenMIPS中的ID模块

只需新增一个接口来接收上一条指令处于执行阶段的运算子类型，接口如下：



## 8.4 测试及结果分析

使用以下代码：



仿真结果为：



结果正确。

观察id模块和ex模块，发现一个疑问：为什么ex模块的stallreq输出端口定义为reg类型，而id模块的stallreq输出端口定义为wire型？