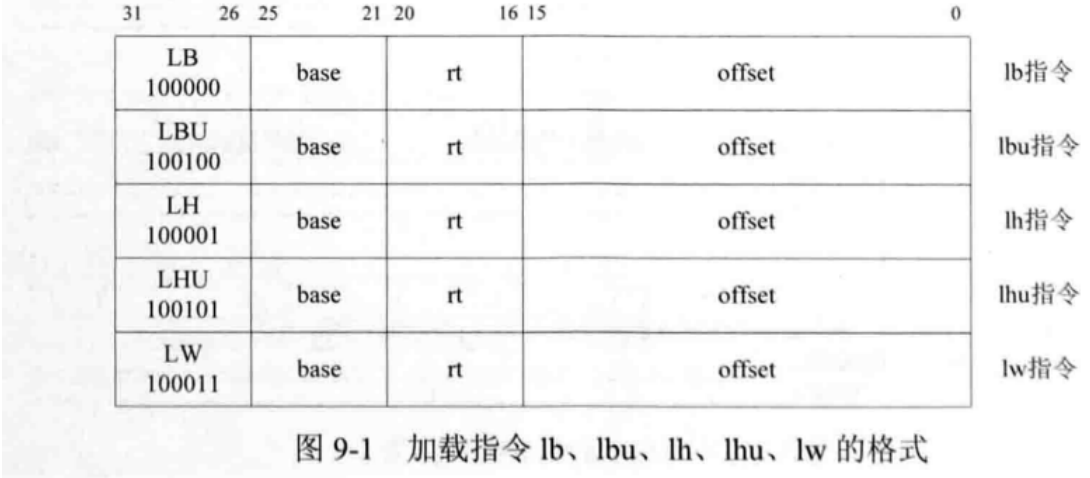
加载存储指令的实现

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。

一、 加载存储指令的说明

1.1 加载指令 lb、lbu、lh、lhu、lw 说明

加载指令 lb、lbu、lh、lhu、lw 的格式如图 9-1 所示。



31 26	25 21	20 16	15 0	
LB 100000	base	rt	offset	lb指令
LBU 100100	base	rt	offset	lbu指令
LH 100001	base	rt	offset	lh指令
LHU 100101	base	rt	offset	lhu指令
LW 100011	base	rt	offset	lw指令

图 9-1 加载指令 lb、lbu、lh、lhu、lw 的格式

我们得到： **加载地址 = signed_extended(offset) + GPR[base]**

其中 offset 是段地址，base 是偏移地址。

指令使用说明：

- 当指令中的指令码为 6'b100000 时，是 lb 指令，字节加载指令。

指令用法为：lb rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个字节，然后符号扩展至 32 位，保存到地址为 rt 的通用寄存器中。

- 当指令中的指令码为 6'b100100 时，是 lbu 指令，无符号字节加载指令。

指令用法为：lbu rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个字节，然后无符号扩展至 32 位，保存到地址为 rt 的通用寄存器中。

- 当指令中的指令码为 6'b100001 时，是 lh 指令，半字加载指令。

指令用法为：lh rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个半字，然后符号扩展至 32 位，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低位为 0。

- 当指令中的指令码为 6'b100101 时，是 lhu 指令，无符号半字加载指令。

指令用法为：lhu rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个半字，然后无符号扩展至 32 位，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低位为 0。

- 当指令中的指令码为 6'b100011 时，是 lw 指令，字加载指令。

指令用法为：lw rt, offset(base)。

指令作用为：从内存中指定的加载地址处，读取一个字，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低两位为 00。

1.2 存储指令 sb、sh、sw 说明

存储指令 sb、sh、sw 的格式如图 9-2 所示。

31 26	25 21	20 16	15 0	
SB 101000	base	rt	offset	sb指令
SH 101001	base	rt	offset	sh指令
SW 101011	base	rt	offset	sw指令

图 9-2 存储指令的格式

我们得到： **加载地址 = signed_extended(offset) + GPR[base]**

下面分别介绍各个存储指令的作用。

- 当指令中的指令码为 6'b101000 时，是 sb 指令，字节存储指令。

指令用法为：sb rt, offset(base)。

指令作用为：将地址为 rt 的通用寄存器的最低字节存储到内存中的指定地址。

- 当指令中的指令码为 6'b101001 时，是 sh 指令，半字存储指令。

指令用法为：sh rt, offset(base)。

指令作用为：将地址为 rt 的通用寄存器的最低两个字节存储到内存中的指定地址。该指令有地址对齐要求，要求计算出来的存储地址的最低位为 0。

- 当指令中的指令码为 6'b101011 时，是 sw 指令，字存储指令。

指令用法为：sw rt, offset(base)。

指令作用为：将地址为 rt 的通用寄存器的值存储到内存中的指定地址。该指令有地址对齐要求，要求计算出来的存储地址的最低两位为 00。

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。

1.3 使用示例

OpenMIPS 处理器是按照字节寻址，并且是大端模式，在这种模式下，数据的高位保存在存储器的低地址中，而数据的低位保存在存储器的高地址中。比如：使用指令 sb 在 0x50 处存储 0x81，存储器中实际存储效果如图 9-3 所示。

使用指令 sh 在 0x54 处存储 0x8281，存储器中实际存储效果如图 9-4 所示。

地址	0x50	0x51	0x52	0x53
数据	0x81	0	0	0

图 9-3 使用指令 sb 在 0x50 处存储 0x81

地址	0x54	0x55	0x56	0x57
数据	0x82	0x81	0	0

图 9-4 使用指令 sh 在 0x54 处存储 0x8281

使用指令 sw 在 0x58 处存储 0x84838281，存储器中实际存储效果如图 9-5 所示。

此时使用加载指令会有如下效果。

地址	0x58	0x59	0x5a	0x5b
数据	0x84	0x83	0x82	0x81

图 9-5 使用指令 sw 在 0x58 处存储 0x84838281

（1）使用指令 lbu 从 0x58 处加载一个字节，读出的字节就是 0x84，经无符号扩展至 32 位是 0x00000084。

（2）使用指令 lb 从 0x58 处加载一个字节，读出的字节就是 0x84，经符号扩展至 32 位是 0xffffff84。

（3）使用指令 lhu 从 0x58 处加载一个半字，读出的半字就是 0x8483，经无符号扩展至 32 位是 0x00008483。

（4）使用指令 lh 从 0x58 处加载一个半字，读出的半字就是 0x8483，经符号扩展至 32 位是 0xffff8483。

（5）使用指令 lh 从 0x59 处加载一个半字，不满足地址对齐要求，会出现异常。

（6）使用指令 lhu 从 0x5a 处加载一个半字，读出的半字就是 0x8281，经无符号扩展至 32 位是 0x00008281。

（7）使用指令 lh 从 0x5a 处加载一个半字，读出的半字就是 0x8281，经符号扩展至 32 位是 0xffff8281。

（8）使用指令 lw 从 0x58 处加载一个字，读出的字就是 0x84838281。

1.4 加载指令 lwl、lwr 说明

加载指令 lwl、lwr 的格式如图 9-6 所示。

31 26	25 21	20 16	15 0	
LWL 100010	base	rt	offset	lwl指令
LWR 100110	base	rt	offset	lwr指令

图 9-6 加载指令 lwl、lwr 的格式

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：

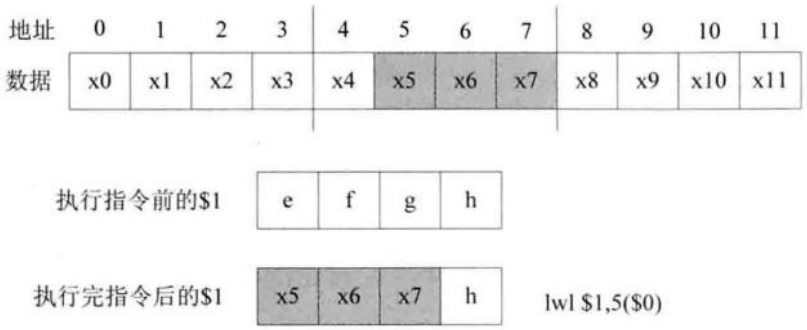


图 9-7 lwl 指令作用举例

lwl 指令执行效果说明：

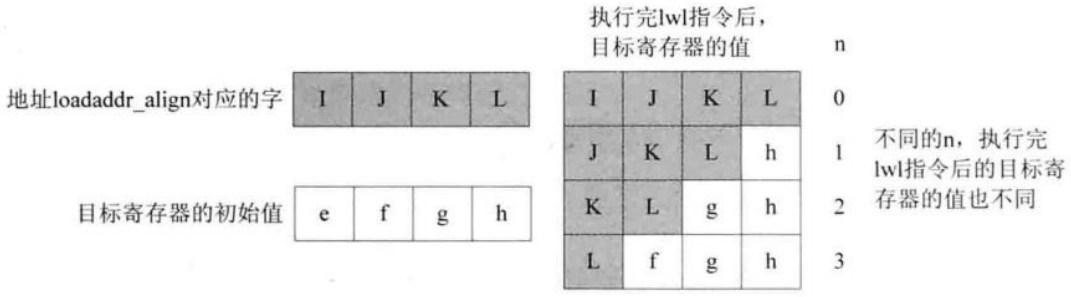


图 9-8 lwl 指令执行效果说明

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：

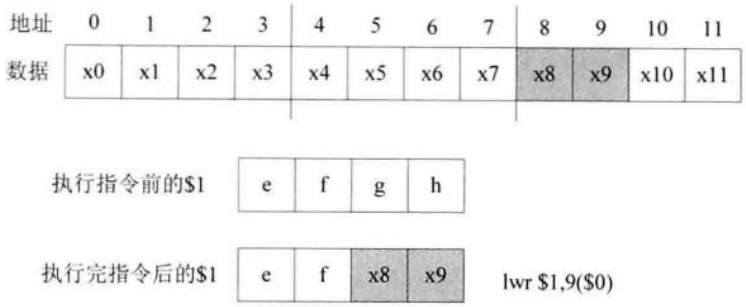


图 9-9 lwr 指令作用举例

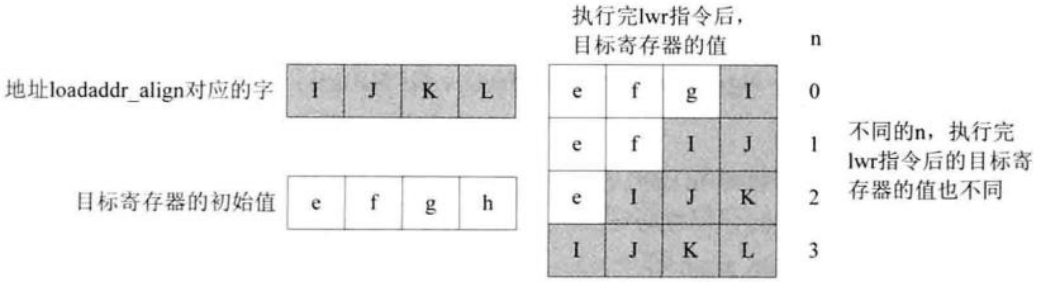


图 9-10 lwr 指令执行效果说明

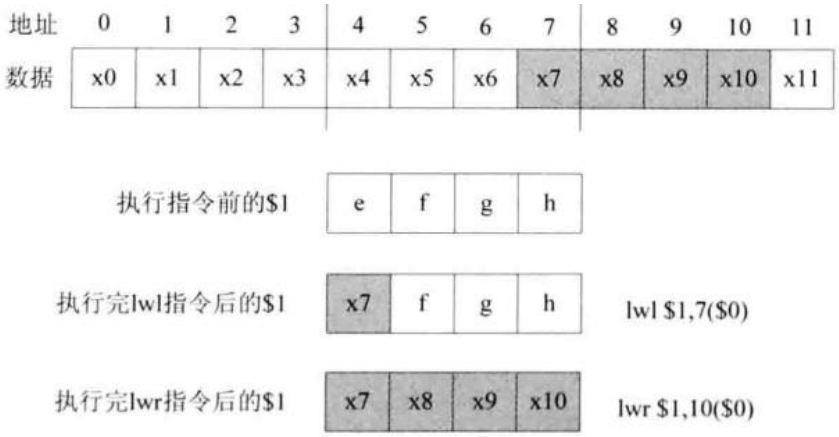
我们通过 lwl 和 lwr 指令配合可以实现从个非对齐地址加载一个字，而且只需要使用 2 条指令就可以实现。比如若要从地址 7 处加载一个字，那么可以使用一下代码来实现，共 5 条指令：

```
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：



1.5 存储指令 swl、swr 说明

格式如下图所示：

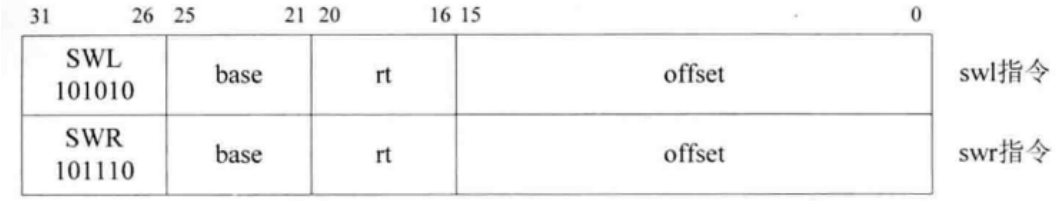


图 9-12 存储指令 swl、swr 的格式

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]

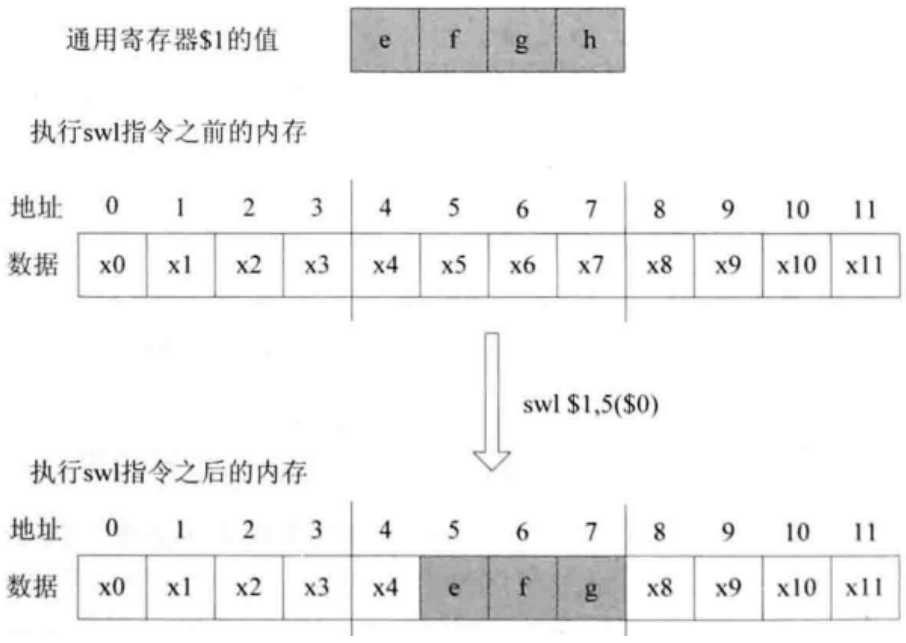
n = storeaddr[1:0]

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：

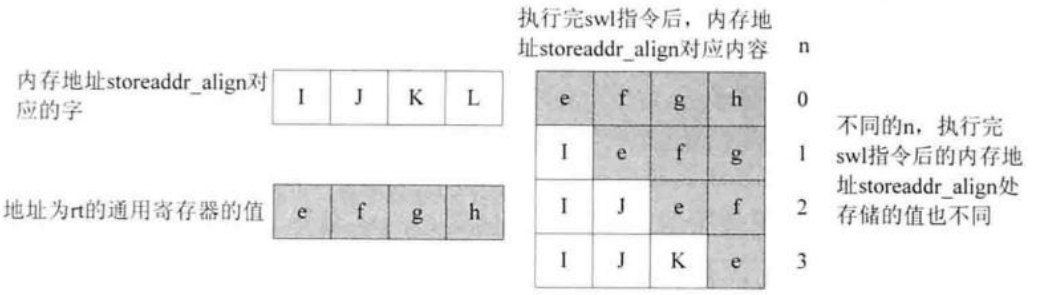


图 9-14　swl 指令执行效果说明

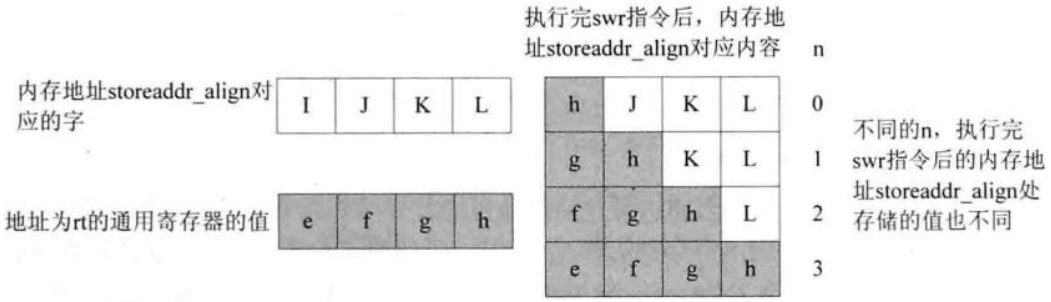


图 9-16　swr 指令执行效果说明

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。

二、 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：

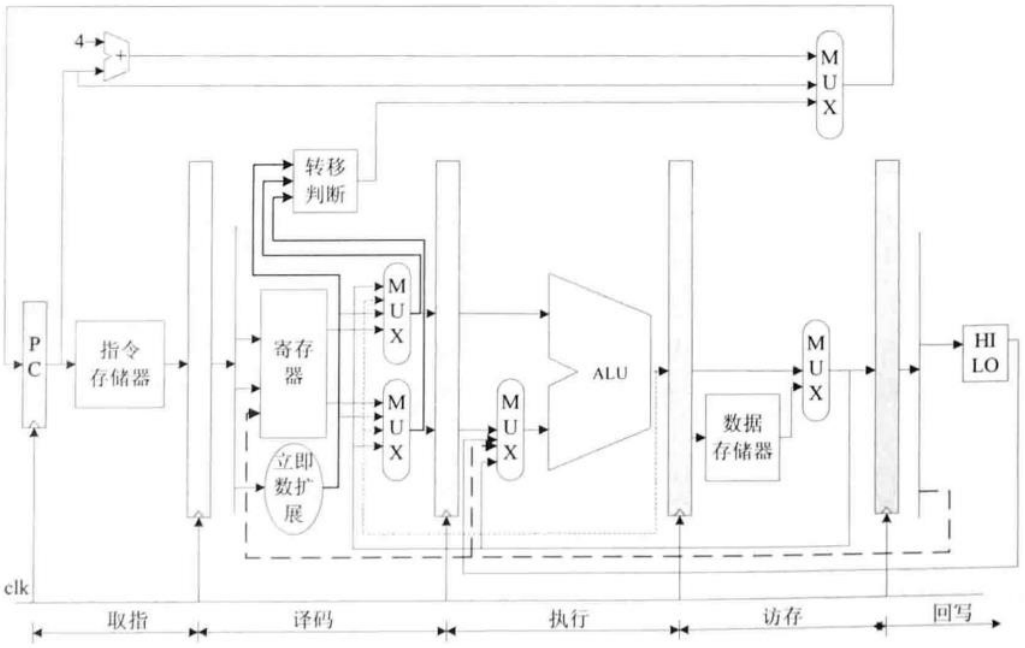


图 9-18 为了实现除 ll、sc 之外的加载存储指令而修改的数据流图

我们也将系统结构进行修改：

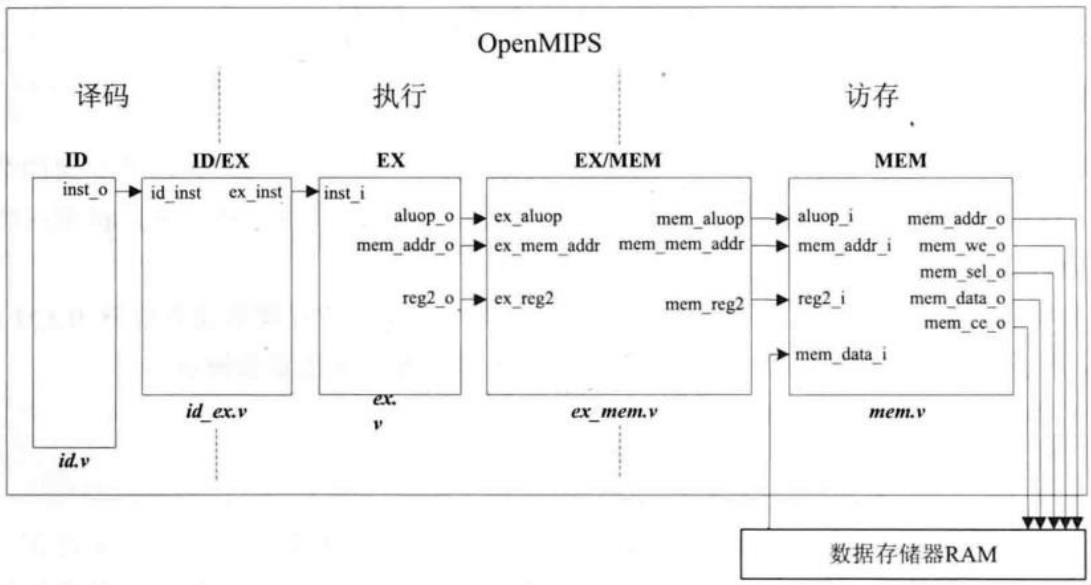


图 9-19 为了实现除 ll、sc 之外的加载存储指令而对系统结构的修改

三、 修改代码具体实现

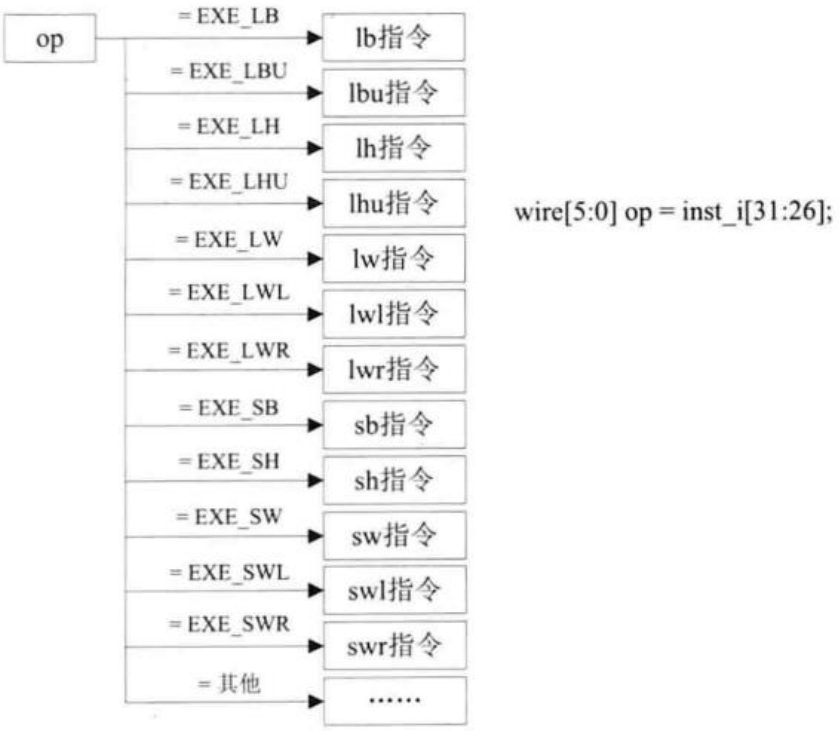
3.1 修改 ID 模块

增加接口如下表：

表 9-1 ID 模块新增加的接口描述

序号	接口名	宽度（bit）	输入/输出	作用
1	inst_o	32	输出	当前处于译码阶段的指令

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：

表 9-2 ID/EX 模块新增加的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	id_inst	32	输入	当前处于译码阶段的指令
2	ex_inst	32	输出	当前处于执行阶段的指令

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：

表 9-3 EX 模块新增加接口的描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	inst_i	32	输入	当前处于执行阶段的指令
2	aluop_o	8	输出	执行阶段的指令要进行的运算子类型
3	mem_addr_o	32	输出	加载、存储指令对应的存储器地址
4	reg2_o	32	输出	存储指令要存储的数据，或者 lwl、lwr 指令要加载到的目的寄存器的原始值

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：

表 9-4 EX/MEM 模块新增加接口的描述

序 号	接 口 名	宽度(bit)	输入/输出	作 用
1	ex_aluop	8	输入	执行阶段的指令要进行的运算的子类型
2	ex_mem_addr	32	输入	执行阶段的加载、存储指令对应的存储器地址
3	ex_reg2	32	输入	执行阶段的存储指令要存储的数据，或者 lwl、lwr 指令要写入的目的寄存器的原始值
4	mem_aluop	8	输出	访存阶段的指令要进行的运算的子类型
5	mem_mem_addr	32	输出	访存阶段的加载、存储指令对应的存储器地址
6	mem_reg2	32	输出	访存阶段的存储指令要存储的数据，或者 lwl、lwr 指令要写入的目的寄存器的原始值

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：

表 9-5 MEM 模块新增加接口的描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	aluop_i	8	输入	访存阶段的指令要进行的运算的子类型
2	mem_addr_i	32	输入	访存阶段的加载、存储指令对应的存储器地址
3	reg2_i	32	输入	访存阶段的存储指令要存储的数据，或者 lwl、lwr 指令要写入的目的寄存器的原始值
4	mem_data_i	32	输入	从数据存储器读取的数据
5	mem_addr_o	32	输出	要访问的数据存储器的地址
6	mem_we_o	1	输出	是否是写操作，为 1 表示是写操作
7	mem_sel_o	4	输出	字节选择信号
8	mem_data_o	32	输出	要写入数据存储器的数据
9	mem_ce_o	1	输出	数据存储器使能信号

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：

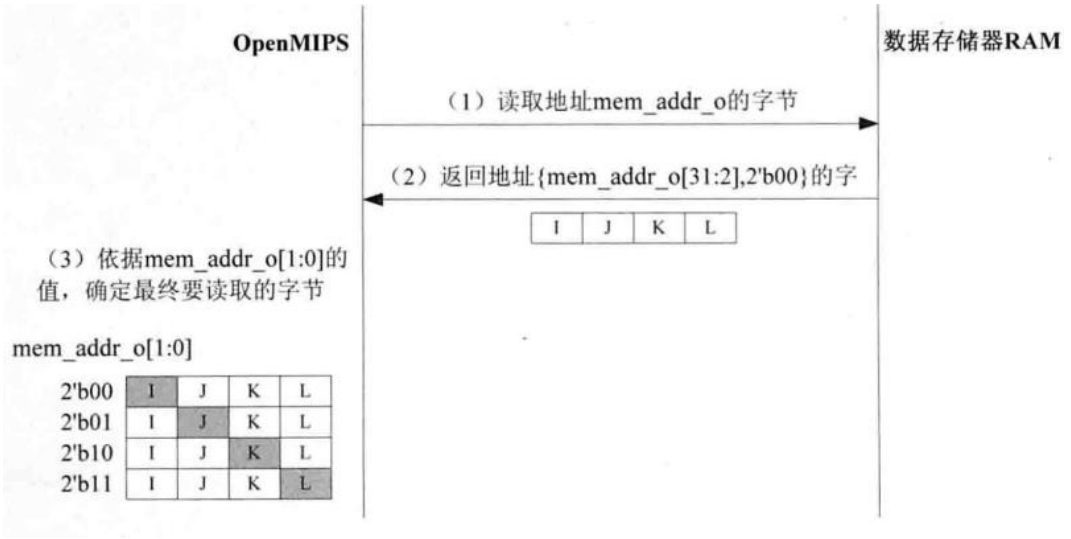
首先对于加载操作，MIPS32 指令集架构可以加载字、半字、字节，但是数据总线宽度是 32 位，只有 4 个字节，因此在 MIPS 架构中 1 个字就是 4 字节。如果是 lb、lh 等加载指令，我们需要知道是加载了一个字中的哪个字节，mem_sel_o 接口就是指出一个字中的有效加载数据是第几字节。举个例子，如果 mem_sel_o 为 4'b0100，表示处理某个字的次高位，也就是 16-23 bit.

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。

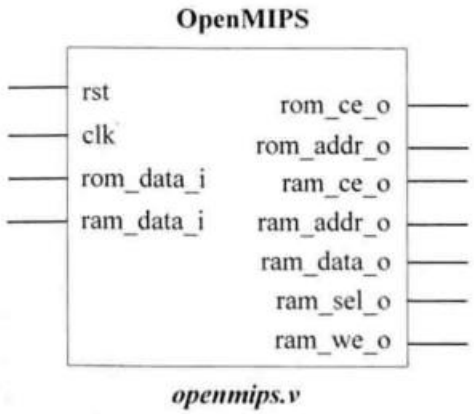
3.6 修改 OpenMIPS 顶层模块

增加的接口描述如下表：

表 9-6 OpenMIPS 模块新增加接口的描述

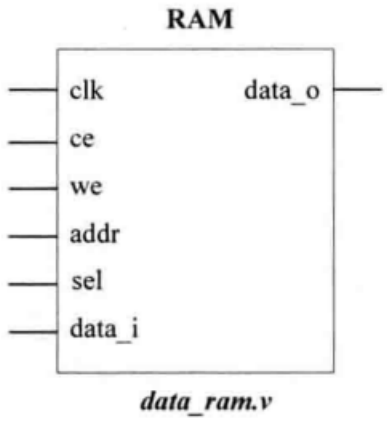
序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	ram_data_i	32	输入	从数据存储器读取的数据
2	ram_addr_o	32	输出	要访问的数据存储器地址
3	ram_we_o	1	输出	是否是对数据存储器的写操作，为 1 表示是写操作
4	ram_sel_o	4	输出	字节选择信号
5	ram_data_o	32	输出	要写入数据存储器的数据
6	ram_ce_o	1	输出	数据存储器使能信号

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：

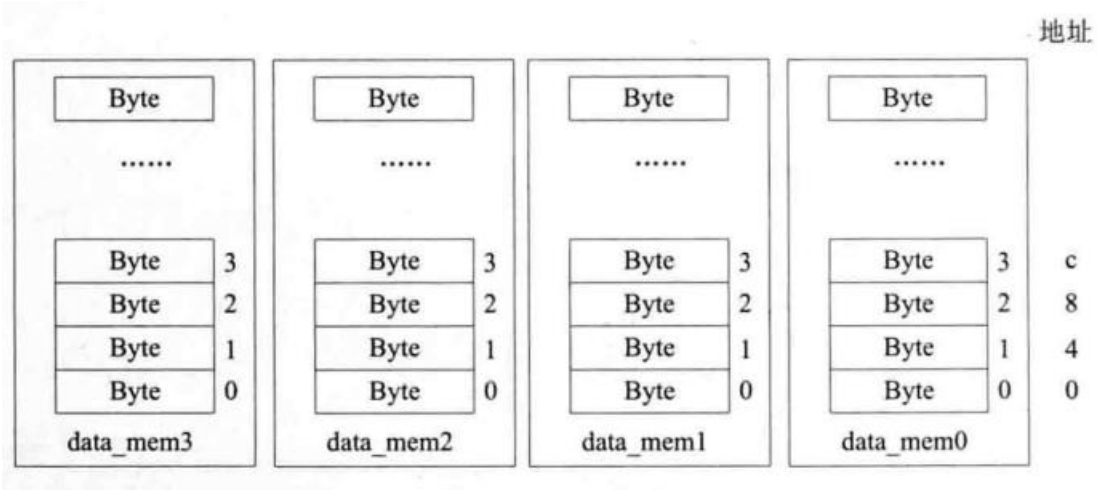
表 9-7 数据存储器 RAM 的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	ce	1	输入	数据存储器使能信号
2	clk	1	输入	时钟信号
3	data_i	32	输入	要写入的数据
4	addr	32	输入	要访问的地址
5	we	1	输入	是否是写操作，为 1 表示是写操作
6	sel	4	输入	字节选择信号
7	data_o	32	输出	读出的数据

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：

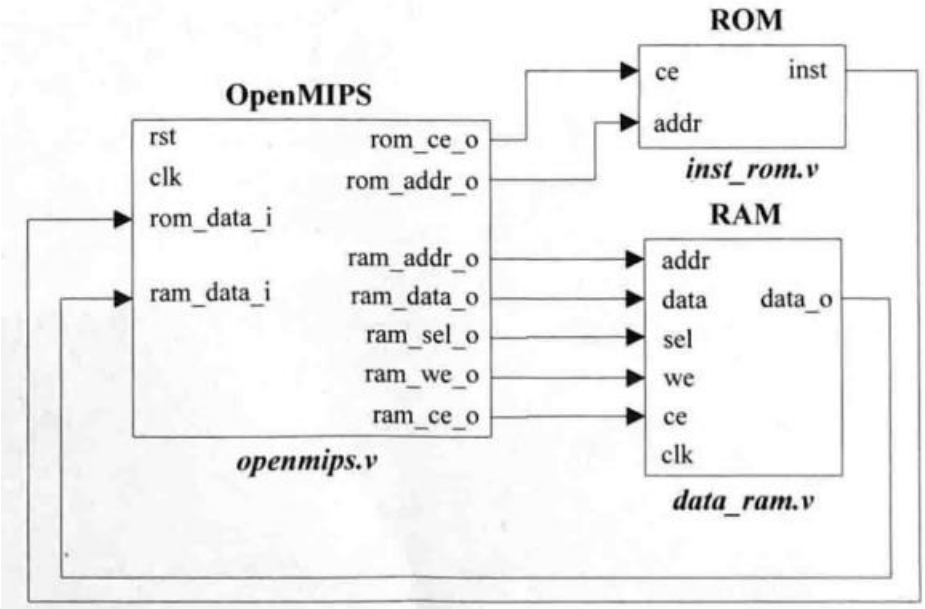


图 9-26 添加数据存储器 RAM 后的最小 SOPC

四、 测试及结果分析

4.1 测试

使用以下测试程序：

```
    .org 0x0
    .set noat
    .set noreorder
    .set nomacro
    .global _start
_start:
    ori   $3,$0,0xeeff
    sb    $3,0x3($0)        # [0x3] = 0xff
    srl   $3,$3,8
    sb    $3,0x2($0)        # [0x2] = 0xee
    ori   $3,$0,0xccdd
    sb    $3,0x1($0)        # [0x1] = 0xdd
    srl   $3,$3,8
    sb    $3,0x0($0)        # [0x0] = 0xcc
    lb    $1,0x3($0)        # $1 = 0xffffffff
    lbu   $1,0x2($0)        # $1 = 0x000000ee
    nop

    ori   $3,$0,0xaabb
    sh    $3,0x4($0)        # [0x4] = 0xaa, [0x5] = 0xbb
    lhu   $1,0x4($0)        # $1 = 0x0000aabb
    lh    $1,0x4($0)        # $1 = 0xffffaabb

    ori   $3,$0,0x8899
    sh    $3,0x6($0)        # [0x6] = 0x88, [0x7] = 0x99
    lh    $1,0x6($0)        # $1 = 0xffff8899
    lhu   $1,0x6($0)        # $1 = 0x00008899

    ori   $3,$0,0x4455
    sll   $3,$3,0x10
    ori   $3,$3,0x6677
    sw    $3,0x8($0)        # [0x8] = 0x44, [0x9]= 0x55, [0xa]= 0x66, [0xb] = 0x77
    lw    $1,0x8($0)        # $1 = 0x44556677


    lwl   $1, 0x5($0)       # $1 = 0xbb889977
    lwr   $1, 0x8($0)       # $1 = 0xbb889944

    nop
    swr   $1, 0x2($0)       # [0x0] = 0x88, [0x1] = 0x99, [0x2]= 0x44, [0x3] = 0xff
    swl   $1, 0x7($0)       # [0x4] = 0xaa, [0x5] = 0xbb, [0x6] = 0x88, [0x7] = 0x44

    lw    $1, 0x0($0)       # $1 = 0x889944ff
    lw    $1, 0x4($0)       # $1 = 0xaabb8844

_loop:
    j _loop
    nop
```

4.2 仿真结果

查看寄存器的值，结果正确。

五、 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。

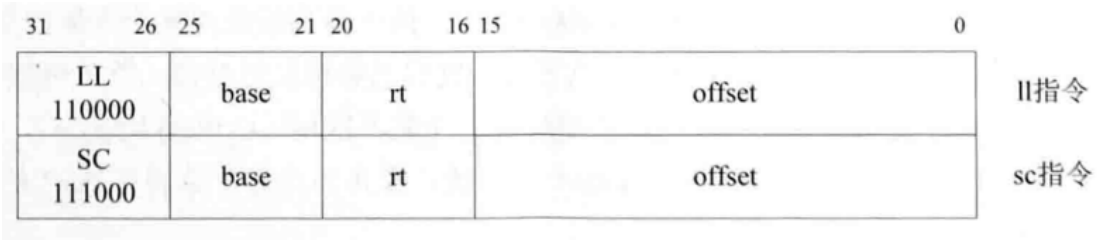
• 多处理器的系统中，另一个 CPU 改写了 RMW 序列要操作的内存空间。

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：



31 26	25 21	20 16	15 0	
LL 110000	base	rt	offset	ll指令
SC 111000	base	rt	offset	sc指令

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：

```
wait:
   ori $1, $0, sem          // sem 是信号量的地址，将这个地址赋给寄存器$1

TryAgain:
   ll  $2, 0($1)              // 获取信号量的值，保存到寄存器$2
   bne $2, $0, WaitForSem     // 如果信号量被占用(其值为 1),那么转移到地址 WaitForSem
                              // 继续等待；如果信号量空闲（其值为 0），那么执行下面的指令

   nop
   ori $2, $0, 1
   sc  $2, 0($1)              // 如果没有被干扰，那么设置信号量被占用（将 1 保存到信号
                              // 量中），同时，设置寄存器$2 为 1，反之，不修改信号量，
                              // 设置寄存器$2 为 0

   beq $2, $0, TryAgain       // 如果寄存器$2 为 0，表示 ll、sc 指令没有成功，未获取到
                              // 信号量，回到 TryAgain 继续尝试
   nop

   jr  $31                    // 反之，表示 ll、sc 指令成功，获取到信号量，可以进入
                              // “临界区域”了。调用 wait 函数时，会将返回地址放在
                              // 寄存器$31，所以此处 jr $31 指令就是回到调用过程，
                              // 进入临界区域
```

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

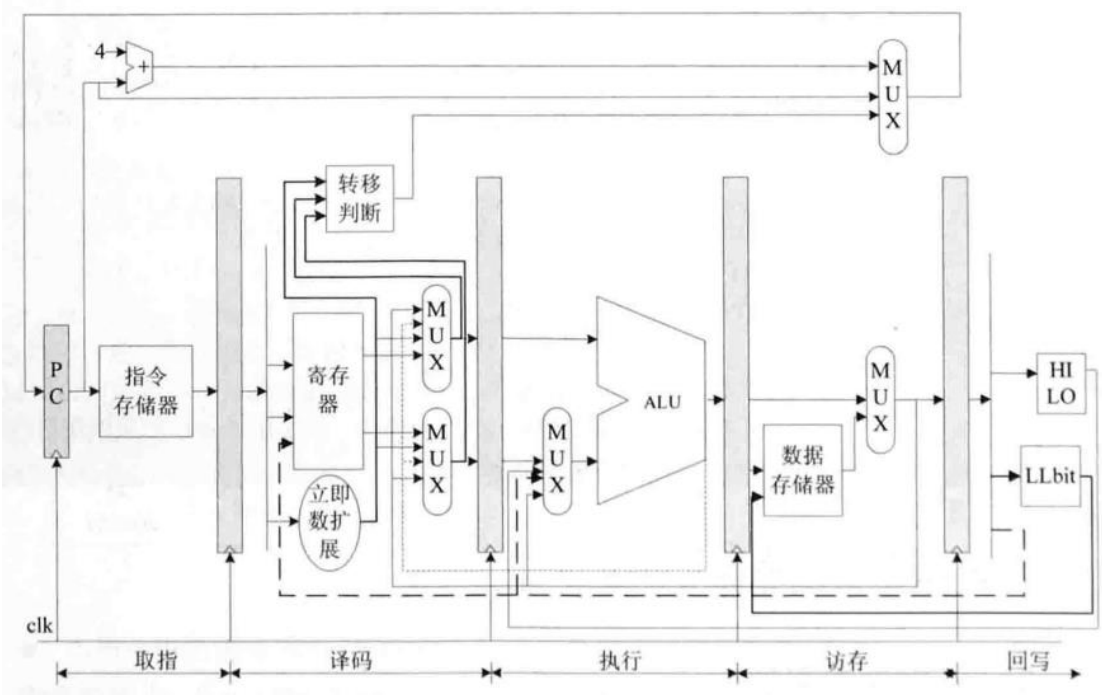


图 9-29 为了实现 ll、sc 指令而修改的数据流图

5.3 系统结构的修改

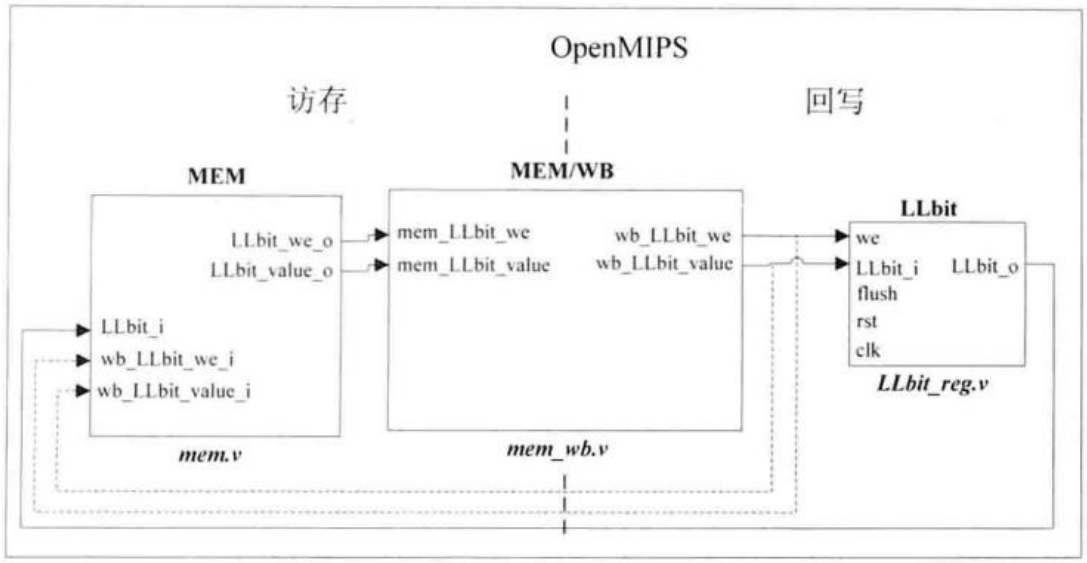


图 9-30 为实现 ll、sc 指令而对系统结构做的修改

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。

六、 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：

表 9-8 LLbit 模块各接口描述

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	flush	1	输入	是否有异常发生
4	we	1	输入	是否要写 LLbit 寄存器
5	LLbit_i	1	输入	要写到 LLbit 寄存器的值
6	LLbit_o	1	输出	LLbit 寄存器的值

```
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

图 9-31 确定 ll、sc 指令的过程

6.3 修改 MEM 模块

接口描述如表所示：

表 9-9 MEM 模块新增接口的描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	LLbit_i	1	输入	LLbit 模块给出的 LLbit 寄存器的值
2	wb_LLbit_we_i	1	输入	回写阶段的指令是否要写 LLbit 寄存器
3	wb_LLbit_value_i	1	输入	回写阶段要写入 LLbit 寄存器的值
4	LLbit_we_o	1	输出	访存阶段的指令是否要写 LLbit 寄存器
5	LLbit_value_o	1	输出	访存阶段的指令要写入 LLbit 寄存器的值

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：

表 9-10 MEM/WB 模块新增接口的描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	mem_LLbit_we	1	输入	访存阶段的指令是否要写 LLbit 寄存器
2	mem_LLbit_value	1	输入	访存阶段的指令要写入 LLbit 寄存器的值
3	wb_LLbit_we	1	输出	回写阶段的指令是否要写 LLbit 寄存器
4	wb_LLbit_value	1	输出	回写阶段的指令要写入 LLbit 寄存器的值

6.5 修改 OpenMIPS 顶层模块

略

七、 测试 ll、sc 指令

使用下面测试代码：

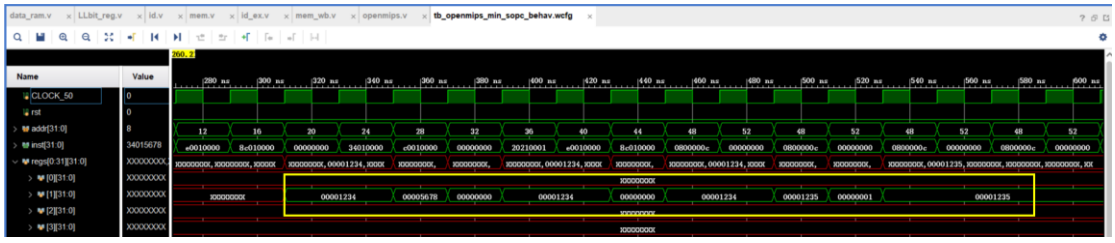
```
        .org 0x0
        .set noat
        .set noreorder
        .set nomacro
        .global _start
_start:
        ori $1,$0,0x1234        # $1 = 0x00001234
        sw  $1,0x0($0)          # [0x0] = 0x00001234

        ori $1,$0,0x5678        # $1 = 0x00005678
        sc  $1,0x0($0)          # $1 = 0x0
        lw  $1,0x0($0)          # $1 = 0x00001234
        nop

        ori $1,$0,0x0           # $1 = 0x0
        ll  $1,0x0($0)          # $1 = 0x00001234
        nop
        addi $1,$1,0x1          # $1 = 0x00001235
        sc  $1,0x0($0)          # $1 = 0x1
        lw  $1,0x0($0)          # $1 = 0x00001235

_loop:
        j _loop
        nop
```

仿真结果为：



结果正确。

八、 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：

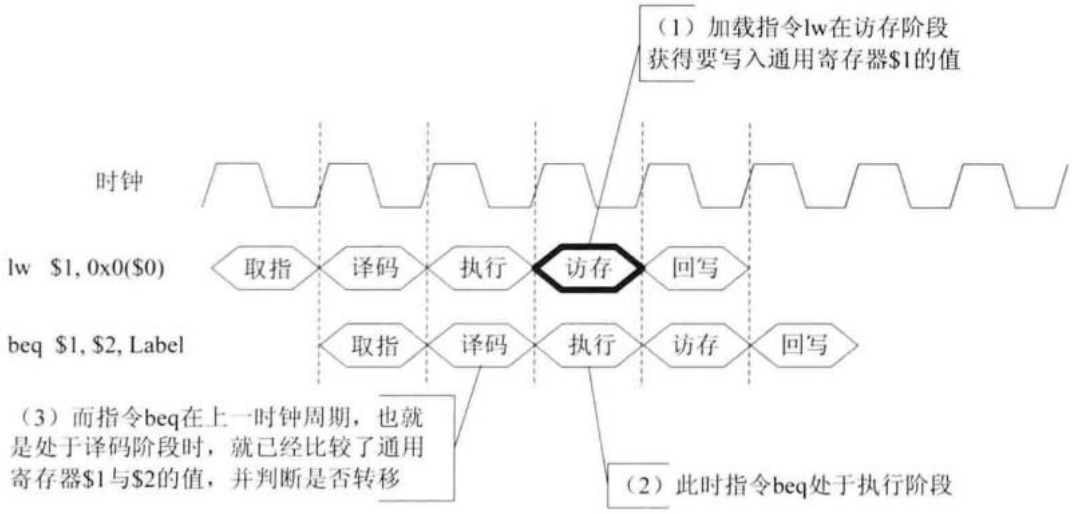


图 9-33 load 相关导致程序执行出错

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：

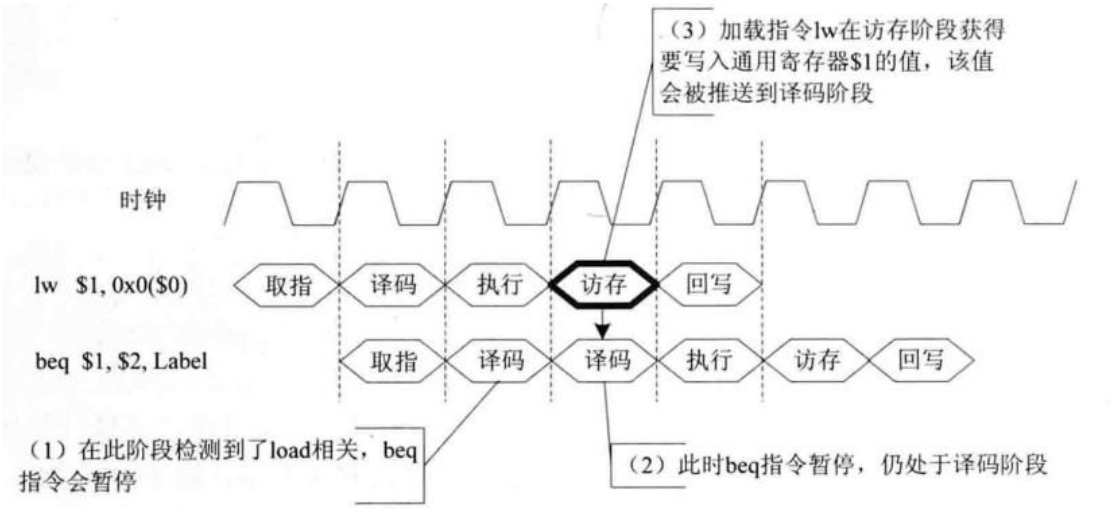


图 9-34 OpenMIPS 对 load 相关的解决方法

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。

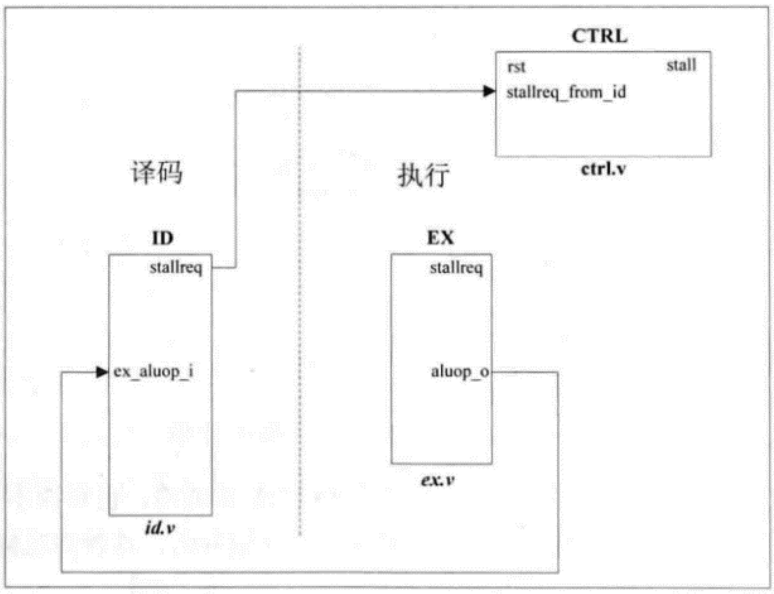


图 9-35 修改 OpenMIPS 系统结构以解决 load 相关问题

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：

表 9-11 ID 模块增加的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	ex_aluop_i	8	输入	处于执行阶段指令的运算子类型

8.4 测试及结果分析

使用以下代码：

```
    .org 0x0
    .set noat
    .set noreorder
    .set nomacro
    .global _start
_start:
    ori $1,$0,0x1234     # $1 = 0x00001234
    sw  $1,0x0($0)       # [0x0] = 0x00001234

    ori $2,$0,0x1234     # $2 = 0x00001234
    ori $1,$0,0x0        # $1 = 0x0
    lw  $1,0x0($0)       # $1 = 0x00001234
    beq $1,$2,Label
    nop

    ori $1,$0,0x4567
    nop

Label:
    ori $1,$0,0x89ab     # $1 = 0x000089ab
    nop

_loop:
    j _loop
    nop
```

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？