乘累加和乘累减指令的实现

一、指令说明

乘累加和乘累减指令包括四条：madd、maddu、msub、msubu，各指令格式如下表所示：

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL2 011100	rs	rt	00000	00000	MADD 000000	madd指令
SPECIAL2 011100	rs	rt	00000	00000	MADDU 000001	maddu指令
SPECIAL2 011100	rs	rt	00000	00000	MSUB 000100	msub指令
SPECIAL2 011100	rs	rt	00000	00000	MSUBU 000101	msubu指令

- 当功能码是 6'b000001 时，表示是 maddu 指令，无符号乘累加运算。

指令用法为：maddu rs, rt。

指令作用为：{HI, LO} <- {HI, LO} + rs × rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为无符号数进行乘法运算，运算结果与{HI, LO}相加，相加的结果保存到{HI, LO}中。

- 当功能码是 6'b000100 时，表示是 msub 指令，有符号乘累减运算。

指令用法为：msub rs, rt。

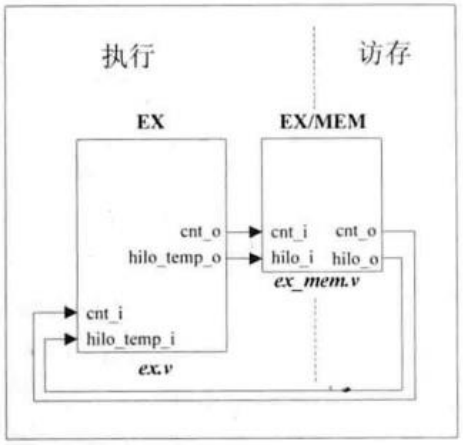
指令作用为：{HI, LO} <- {HI, LO} - rs × rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为有符号数进行乘法运算。然后使用{HI, LO}减去乘法结果，相减的结果保存到{HI, LO}中。

- 当功能码是 6'b000101 时，表示是 msubu 指令，无符号乘累减运算。

指令用法为：msubu rs, rt。

指令作用为：{HI, LO} <- {HI, LO} - rs × rt，将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值作为无符号数进行乘法运算。然后使用{HI, LO}减去乘法结果，相减的结果保存到{HI, LO}中。

二、实现思路



我们在之前将 CPU 增加了流水线暂停机制，因此我们计划在流水线执行阶段采用两个时钟周期完成运算，第一个时钟周期执行乘法运算，第二个周期将乘法结果与 HILO 寄存器的结果相加/相减。

要实现乘累加和乘累减，要保存两个信息：当前的时钟周期，以及乘法结果。因此我们在 EX/MEM 模块中添加两个寄存器 cnt、hilo，分别保存上述信息。修改后的系统结构如图所示：

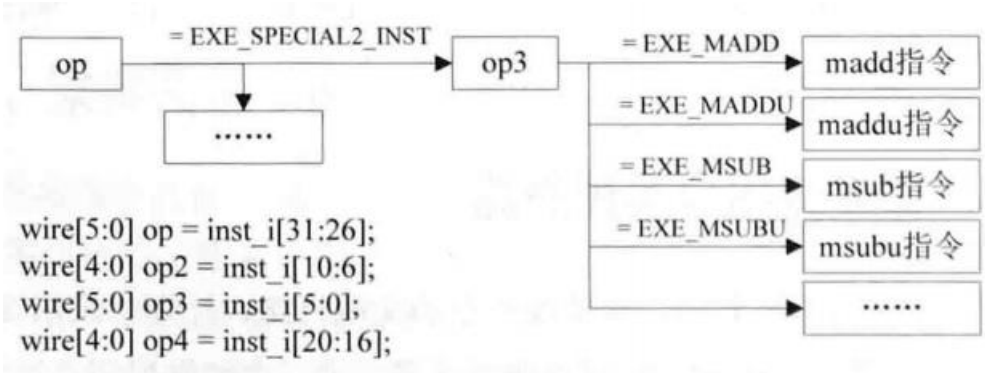
执行阶段 EX 模块的输出 hilo_temp_o 是乘法结果，传递到 EX/MEM 模块，并在下一个时钟周期送回 EX 模块，参与第二个时钟周期的加法/减法运算。

执行阶段 EX 模块的输出 cnt_o 代表当前是第几个时钟周期，传递到 EX/MEM 模块，并在下一个时钟周期送回 EX 模块，后者据此判断当前处于乘累加、乘累减指令的第几个执行周期。

三、修改实现

3.1 修改 ID 模块

确定指令过程如图：



具体代码见**附录一**

值得注意的是：因为最终将结果是写入 HILO 寄存器，而不是写入通用寄存器，所以设置 wreg_o 为 WriteDisable。但是我发现了一个问题，将 MADD 和 MADDU 类型设置成 MUL 类型，在执行阶段这是保存到通用寄存器的操作，为什么这里 MUL 可以执行保存到 HILO 寄存器呢？

往后继续学习可能会有答案。

3.2 修改 EX 模块

增加的接口描述如下表所示：

表 7-2 EX 模块增加的接口描述

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	hilo_temp_i	64	输入	第一个执行周期得到的乘法结果
2	cnt_i	2	输入	当前处于执行阶段的第几个时钟周期
3	hilo_temp_o	64	输出	第一个执行周期得到的乘法结果
4	cnt_o	2	输出	下一个时钟周期处于执行阶段的第几个时钟周期

具体代码见**附录二**

我们首先计算乘法结果，取出乘法操作的被乘数，指令madd、msub都是有符号乘法，我们用$\sim reg1_i + 1$进行正负数转换，，同理第二个数也是一样，目的是全部转换成乘法运算，再根据两个乘数的符号来判断最后结果的正负。

然后我们得到了临时乘法结果hilo_temp，根据以下的几种情况来判断正负：

（1） 若为有符号乘法madd、msub

a) 乘数和被乘数为一正一负，则对最终结果hilo_temp求补码；

b) 若两者同号，则最终结果不变；

（2） 若为无符号乘法运算，最终结果也不变

另外，在乘累加乘累减指令中的流水暂停中，我们引入了新的变量cnt，用来记录周期数，当处于第一个周期时，将stallreq_for_madd_msub设置为Stop，当进入第二个周期时，此时 EX 模块驶入的hilo_temp_i就是上一个时钟周期得到的乘法结果，所以将hilo_temp_i与HILO寄存器相加，保存到hilo_temp1中，同时设置stallreq_for_madd_msub为NoStop，乘累加指令结束，设置cnt_o为2’b10，而不是直接设置为2’b00，目的是，如果是其他原因导致流水线保持暂停，那么由于cnt_o为2’b10，所以 EX 阶段不在计算，从而方式乘累加指令重复运行。

最后给出信号stallreq的值，目前只有乘累加乘累减指令会导致流水线暂停，所以stallreq就直接等于变量stallreq_for_madd_msub的值。

3.3 修改 EX/MEM 模块

增加的接口如下表所示：

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	hilo _i	64	输入	保存的乘法结果
2	cnt_i	2	输入	下一个时钟周期是执行阶段的第几个时钟周期
3	hilo_o	64	输出	保存的乘法结果
4	cnt_o	2	输出	当前处于执行阶段的第几个时钟周期

具体代码见**附录三**

3.4 修改 OpenMIPS 模块

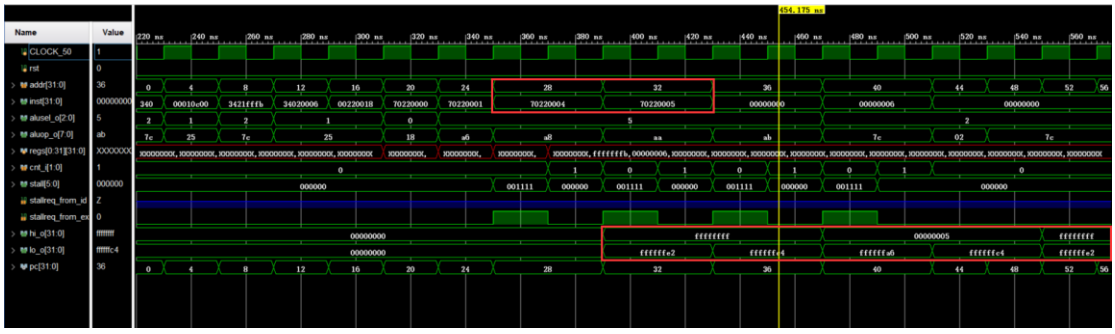
这部分没什么好说的，就是细心一点，变量的值不要弄混淆了，类似于.cnt_o接口的值是cnt_i等等，具体代码见**附录四**。

四、测试及结果

我使用了下面的测试程序进行测试：

```
        .org 0x0
        .set noat
        .global _start
    _start:
        ori   $1,$0,0xffff
        sll   $1,$1,16
        ori   $1,$1,0xfffb          # $1 = -5
        ori   $2,$0,6               # $2 = 6

        mult $1,$2                  # hi = 0xffffffff
                                    # lo = 0xfffffffe2

        madd $1,$2                  # hi = 0xffffffff
                                    # lo = 0xffffffc4

        maddu $1,$2                 # hi = 0x5
                                    # lo = 0xffffffa6

        msub $1,$2                  # hi = 0x5
                                    # lo = 0xffffffc4

        msubu $1,$2                 # hi = 0xffffffff
                                    # lo = 0xfffffffe2
```

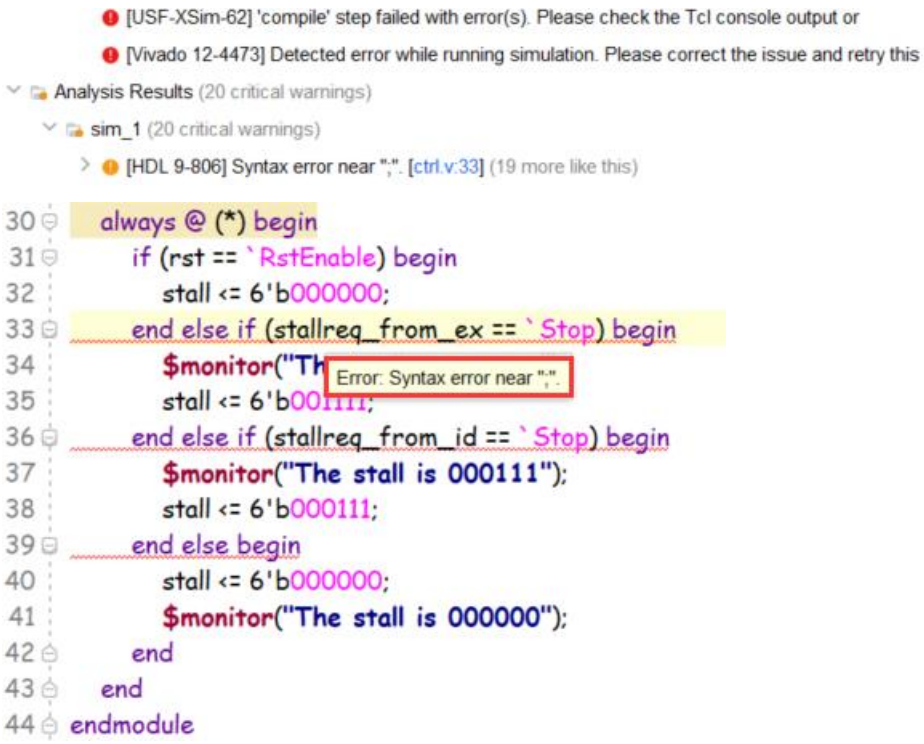
最终仿真结果为：



结果正确。

另外，我在给实验班编了几个错误中，包含了一些需要理解源程序以及整个工作原理才能找到的错误（当然一行行对照源码也能改出来）。

（1） 首先是一个语法错误，就是在 define 宏定义后面加上了分号，这是在预处理的时候就执行的语句，本质上并不是一个语句，所以不能加分号，最终加上分号的结果是编译错误，提示这样的信息：

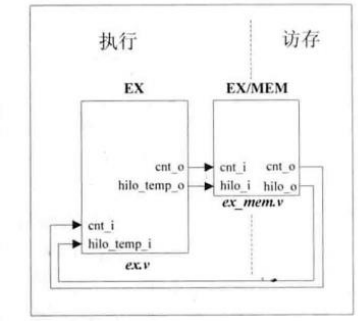


只是提示在分号处有语法错误，这样其实很难发现是宏定义出错~~~

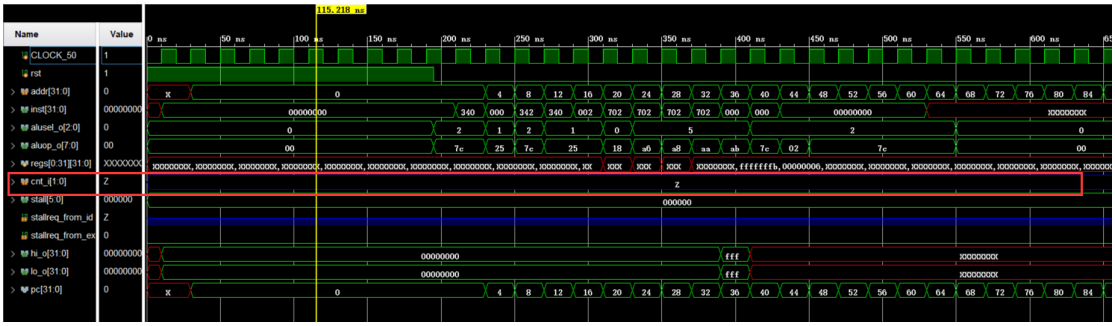
（2） 然后就是在 OpenMIPS 模块中实例化参数错误：

```
241   //EX/MEM module for instance
242   ex_mem ex_mem0(
243     .clk(clk),
244     .rst(rst),
245
246     .cnt_i(cnt_o),
247     .hilo_i(hilo_temp_o),
248     .stall(stall),
249
250     //Message sent from EX module
251     .ex_wd(ex_wd_o),
252     .ex_wreg(ex_wreg_o),
253     .ex_wdata(ex_wdata_o),
254     .ex_whilo(ex_whilo_o),
255     .ex_hi(ex_hi_o),
256     .ex_lo(ex_lo_o),
257
258     //Message sent to MEM module
259     .mem_wd(mem_wd_i),
260     .mem_wreg(mem_wreg_i),
261     .mem_wdata(mem_wdata_i),
262     .mem_whilo(mem_whilo_i),
263     .mem_hi(mem_hi_i),
264     .mem_lo(mem_lo_i),
265     .cnt_o(cnt_i),
266     .hilo_o(hilo_temp_i)
267   );
```

我将这两个参数互换了一下，如果熟悉右边这张系统模块图的话，根据最终仿真结果cnt为高阻态，很容易知道是接口实例化出错。看右边的图我们知道，EX 模块的cnt_i和 EX/MEM 模块的cnt_o的值应该是一样的，而变量cnt_o是 EX 模块的cnt_o接口的值，它应该和 EX/MEM 模块中cnt_i的值一致。检查接口实例化，应该会发现 EX/MEM 模块中那两个cnt并没有正确传值；



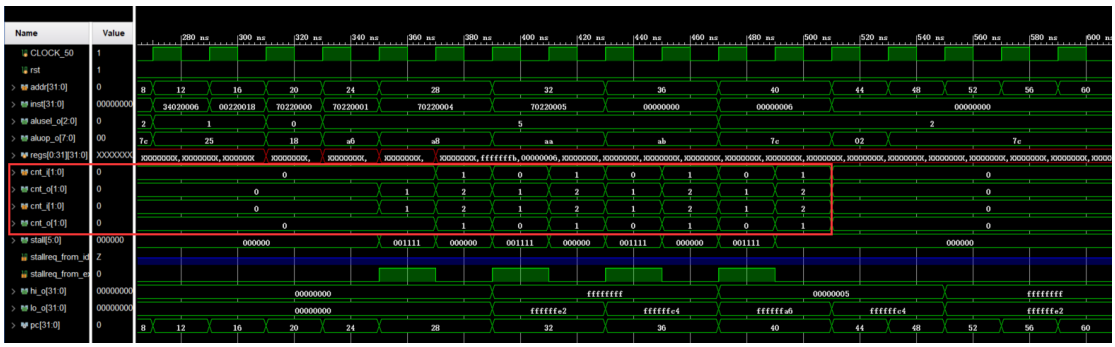
仿真发现cnt为高阻态：



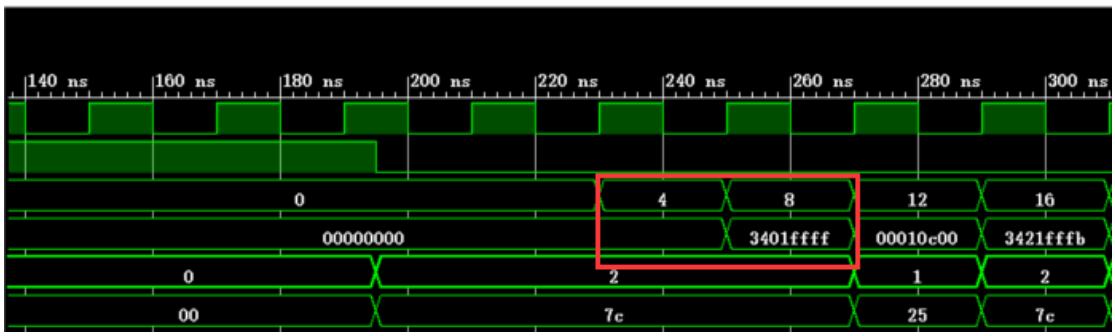
（3） 在id模块中，我用assign自行赋值了stallreq，最终仿真结果的stallreq_from_id便为我赋值的值，正确的应为高阻态（整个程序并为对其赋值）

（4） 还有就是在 EX/MEM 模块中一段代码写错，当stall[3]为 0 时，cnt_o 应为2’b00，我将它改为了cnt_i，这样 EX 模块中的cnt_i的值便一直为2’b10，程序便会在一次流水暂停后一直暂停下去，不会跳出暂停；

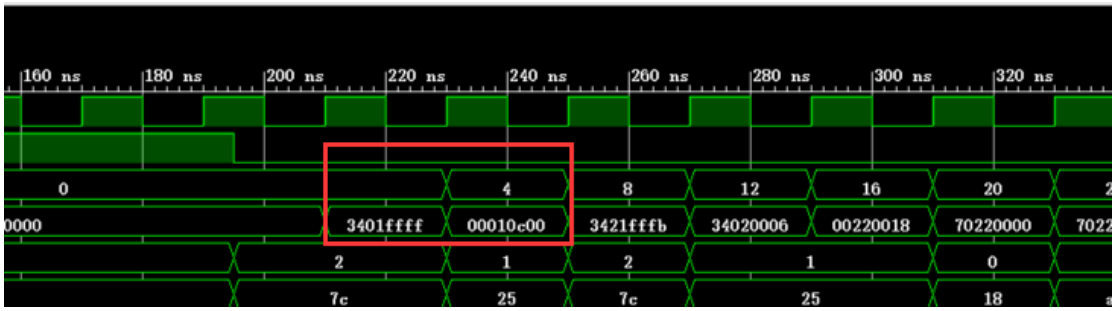
正确的应为：



（5） 当然还有一个最难发现的错误，就是汇编代码的一个小问题，我将第一行的.org 0x0改为了.org 0x8，结果虽然正确，但是细心的人会发现，读取指令延后了两个周期：



修改之前正确的应为：



当然还有一些没有意义的改错，譬如故意将aluop之类的指令码写错，少一个end之类的，就没有出的必要了。我觉得能从错误中进一步理解代码和流程才是改错的核心所在。

五、附录

不写了，懒得贴代码了。