# 实验十一 实现乘除法指令的理想流水线设计实验

## 实验目的

1. 掌握乘法指令和除法指令的数据通路。
2. 掌握在执行单元中添加乘除法运算部件 MDU 的方法。
3. 掌握在五级流水线中实现乘法指令和除法指令的方法。

## 实验原理与实验内容

### RV64M拓展指令集

RISC-V的乘除法指令被包含在M拓展指令集内，RV64I中并不包含任何乘除法指令。不过，即使处理器未实现乘除法，也能运行完整的RISC-V软件栈，这可以减小嵌入式芯片的面积。需要注意的是，M拓展指令集中乘除指令的操作数均为整数。

* + - * 1. RV64M拓展指令

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 |  | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 |  |
| 0000001 | | | rs2 | | rs1 | | 000 | | rd | | 0110011 | | MUL |
| 0000001 | | | rs2 | | rs1 | | 001 | | rd | | 0110011 | | MULH |
| 0000001 | | | rs2 | | rs1 | | 010 | | rd | | 0110011 | | MULHSU |
| 0000001 | | | rs2 | | rs1 | | 011 | | rd | | 0110011 | | MULHU |
| 0000001 | | | rs2 | | rs1 | | 100 | | rd | | 0110011 | | DIV |
| 0000001 | | | rs2 | | rs1 | | 101 | | rd | | 0110011 | | DIVU |
| 0000001 | | | rs2 | | rs1 | | 110 | | rd | | 0110011 | | REM |
| 0000001 | | | rs2 | | rs1 | | 111 | | rd | | 0110011 | | REMU |
| 0000001 | | | rs2 | | rs1 | | 000 | | rd | | 0111011 | | MULW |
| 0000001 | | | rs2 | | rs1 | | 100 | | rd | | 0111011 | | DIVW |
| 0000001 | | | rs2 | | rs1 | | 101 | | rd | | 0111011 | | DIVUW |
| 0000001 | | | rs2 | | rs1 | | 110 | | rd | | 0111011 | | REMW |
| 0000001 | | | rs2 | | rs1 | | 111 | | rd | | 0111011 | | REMUW |

RV64M包含有符号整数除法指令（DIV）和无符号整数除法指令（DIVU），它们将商写入目的寄存器。但少数时候程序员需要余数而不是商，因此RV64M提供求有符号余数指令（REM）和求无符号余数指令（REMU），它们将余数写入目的寄存器。同时为了支持32位长度的数据，RV64M中还有以W结尾的字指令。

除法的运算公式如下，其中rs1为被除数、rs2为除数：

或

RISC-V提供了4种类型的除法指令：

1. 用于获取商的指令有2种，商的符号类型和运算类型相对应：
   1. 两个有符号数运算时使用DIV。
   2. 两个无符号数运算时使用DIVU。
2. 用于获取余数的指令有2种，余数的符号与被除数（rs1）相同：
   1. 两个有符号数运算时使用REM。
   2. 两个无符号数运算时使用REMU。

乘法的运算公式如下，其中rs1为被乘数、rs2为乘数：

相比于除法，在计算过程中乘法的位数最大会达到128位，因为积的位宽是被乘数和乘数两者位宽之和。RISC-V提供了4种类型的乘法指令：

1. MUL指令用于获取积的低64位。
2. 想要得到积的高64位，根据源操作数的符号类型分又有3种情况，其中的H表示High，即取结果的高位：
   1. 两个有符号数相乘时使用MULH。
   2. 两个无符号数相乘时使用MULHU。
   3. rs1有符号rs2无符号时使用MULHSU。

用一条指令将128位积写入两个64位寄存器会增加硬件复杂度，因此RV64M需要两条乘法指令才能得到完整的128位积。

在几乎所有的处理器上在几乎所有处理器上，乘法的运算速度比移位和加法都慢，而除法又比乘法慢得多。编译器一般会先对乘除法进行优化，将2的幂次类的乘除法转换为左右移位的操作。在RISC-V指令集手册第一卷[[1]](#footnote-1)的7.1节举例了一种指令融合的操作。对于乘法指令而言：先使用MULH[[S]U]指令[[2]](#footnote-2)紧接着使用MUL指令，像这样的指令序列可以通过微架构设计加速MUL指令的计算[[3]](#footnote-3)。除法也有类似的指令融合操作，可以参考上述手册的7.2节。指令融合体现了一种软硬件结合的思想。

* + - * 1. 除法发生除以0或溢出时结果的取值

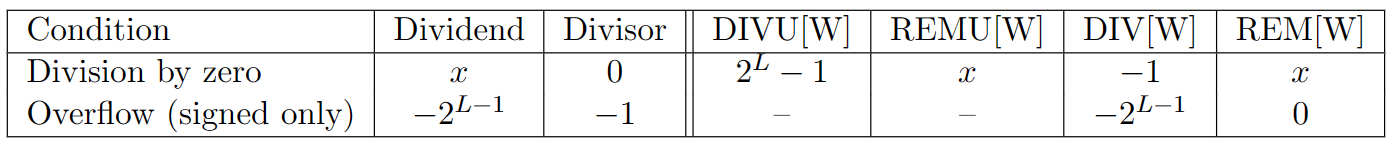


表1-2展示了除法发生除以0或溢出时结果的取值。L表示运算数的宽度，对于DIV[U]和REM[U]而言L为64，对于DIV[U]W和REM[U]W而言L为32。

### 数据通路设计

相比于除法指令，乘法指令在运算时的位数更多更加复杂，因此选取乘法指令作为例子。在乘法指令中，MUL只使用128位积的低64位，实际上并不用考虑源操作数的符号类型，因此这应该是乘法指令中最容易实现的。而在乘法取高位的指令中MULHSU既包含了有符号操作数又包含了无符号操作数难度应该是最大的，因此这里以MULHSU作为例子讲述数据通路的设计。

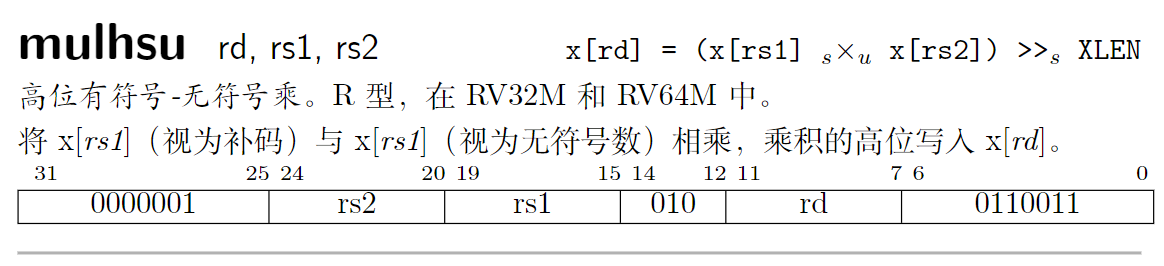
前端

前端只负责指令的准备，指令的执行细节都在后端部分，因此无需修改前端的数据通路。

后端

前端部分已经成功取得指令，经过前面几个实验的练习应该了解了指令在后端中执行的一般流程：先通过译码识别出这条指令为MULHSU指令，并产生相应的控制信号，之后指令将在执行单元中完成运算，最后在写回单元里写回通用寄存器堆。

##### 译码单元



* + - 1. MULHSU指令定义

观察表1-1里的指令格式我们可以知道，M拓展指令集中的指令都是R型指令，也就是两个源操作数均来自于通用寄存器堆。而R型指令在译码单元准备源操作数的数据通路在实验九中就已经设计完成，因此这点无需进行修改。图1-1展示了MULHSU的指令定义。

下面需要考虑译码器译码得到的信号是否够用。

目前译码器译码得到的info数据包内信号包含src1\_ren、src1\_raddr、src2\_ren、src2\_raddr、op、reg\_wen和reg\_waddr。ALU只负责单周期的简单整数运算，对于乘除法指令需要在执行单元中增加一个新的运算部件——MDU（Multiplication-Division Unit）也就是乘除法单元。

为了能够在执行单元中选择运算部件，译码信号也需要相应的增加。这里增加了一个fusel信号，用于选择功能单元。对于所有在ALU中执行的指令，可以将fusel编码为0；对于乘除法这些需要在MDU中执行的指令，可以将fusel编码为1。

在chisel中可以新建一个object类专门存放fusel的类型，这样有利于统一管理避免硬编程行为的出现。代码如下：

object FuType {

  def num     = 2

  def alu     = "b0".U // arithmetic logic unit

  def mdu     = "b1".U // multiplication division unit

  def apply() = UInt(log2Up(num).W)

}

指令在执行单元中将先通过fusel选择相应的功能单元，功能单元再按照op执行相应的运算操作。观察表1-1的func3字段可以发现，乘法指令和除法指令可以使用func3的最高位也就是指令的14位进行区分，指令的14位为0是乘法，为1是除法。字指令与非字指令只有opcode段有区别，因此对于op的编码可以直接选用指令的func3字段，并额外在高位增加一位用于区分是否为字指令。MULHSU的func3字段值为010且该指令不为字指令，因此其op可以设计为0010；同理，DIVUW的op可以设计为1101。

在chisel中可以新建一个object类专门存放MDU中op的定义，并且可以定义相应的函数方便后续使用。代码如下：

object MDUOpType {

  def mulhsu = "b0010".U

  def divuw = "b1101".U

  def isDiv(op:     UInt) = op(2)

  def isDivSign(op: UInt) = isDiv(op) && !op(0)

  def isWordOp(op:       UInt) = op(3)

}

至此便完成了译码单元数据通路的升级，译码器需要产生的信号如表1-3所示。设计完成的译码单元结构如图1-1所示。对于info数据包定义的chisel代码如下：

class Info extends Bundle {

  val src1\_ren   = Bool()

  val src1\_raddr = UInt(REG\_ADDR\_WID.W)

  val src2\_ren   = Bool()

  val src2\_raddr = UInt(REG\_ADDR\_WID.W)

  val fusel      = FuType()

  val op         = FuOpType()

  val reg\_wen    = Bool()

  val reg\_waddr  = UInt(REG\_ADDR\_WID.W)

  val imm        = UInt(XLEN.W)

  val inst       = UInt(XLEN.W)

}

可以发现虽然增加了fusel这一个信号但是并未对结构图产生影响，这是因为所有的控制信号被封装在info数据包中，只要对info这一个bundle的内容进行修改，所有以info实例化的信号量均会发生变化，chisel的这种面向对象的设计语言使得信号修改不必牵一发而动全身了。

* + - * 1. 译码信号

| 信号名 | 含义 |
| --- | --- |
|  | 已经实现的信号 |
| inst | 32位指令 |
| src1\_ren  src1\_raddr  src2\_ren  src2\_raddr  op  reg\_wen  reg\_waddr | src1是否需要读通用寄存器堆  src1的通用寄存器堆读地址  src2是否需要读通用寄存器堆  src2的通用寄存器堆读地址  指令的操作类型  是否需要写回通用寄存器堆  通用寄存器堆的写地址 |
| imm | 立即数 |
|  | 需要增加的信号 |
| fusel | FU[[4]](#footnote-4)选择信号 |



* + - 1. 译码单元

##### 执行级缓存

缓存级数据包如下：

class IdExeData extends Bundle {

  val pc       = UInt(XLEN.W)

  val info     = new Info()

  val src\_info = new SrcInfo()

}

info数据包被包含于data数据包内，因此info的修改同样会自动修改data的内容，不必对执行级缓存进行手动修改。访存级缓存和写回级缓存也是一样的道理，均无需进行修正。

##### 执行单元

1. 总体思路

执行单元负责指令结果的运算。在高性能的CPU中，乘法器和除法器被设计为需要执行多个周期的部件，这样才有利于CPU主频的提升。不过对于实验而言，CPU的性能并不是主要关注对象，功能的实现才是最重要的目标。因此，乘法器与除法器均实现为单周期，即当拍出结果。对于实验结果的判定，只要计算结果一致，就认定设计正确，并不考虑性能高低。综上所述，执行单元设计结构如图1-2所示。

src\_info和info.op被传入ALU还有MDU中的MUL和DIV。根据info.op的第二位选择来自于MUL或DIV的计算结果，也就是区分乘除法。

这里我们对rd\_info的定义进行了修改，rd\_info的wdata被定义为向量类型，其包含了FuType.num个同类型的UInt数据，此处FuType.num的值就是先前定义的2。

class RdInfo extends Bundle {

  val wdata = Vec(FuType.num, UInt(XLEN.W))

}

ALU的计算结果给rd\_info.wdata(FuType.alu)赋值，这里的FuType.alu的值就是在对象FuType中定义的“b0”也就是0。MDU的计算结果给rd\_info.wdata(FuType.mdu)赋值，也就是rd\_info.wdata(1)。这样子设计需要最后在写回单元中根据info.fusel的值选择rd\_info中的一个wdata。

当然也可以不按照这种思路，在执行单元内就直接根据fusel将rd\_info的wdata选择完毕。



* + - 1. 执行单元

1. 关于乘法器的设计

64位有/无符号乘法可以转化为65位有符号乘法，即将64位有/无符号数转化为65位有符号数：对64位有符号数，最高位扩展为符号位；对64位无符号数，最高位补0。

因此，在源操作数输入乘法器时，根据指令需求进行源操作数的拓展，对于MULHSU而言，其被乘数是有符号数，乘数是无符号数，则需要对rs1\_data的65位填充和64位一致的值作为被乘数，对rs2\_data的65位填充0作为乘数，在乘法器内只需要将被乘数乘以乘数即可 ，最终的低128位为乘法器输出结果，再截取高64位作为MULHSU的结果。

##### 访存级缓存和访存单元

无需修改。

##### 写回单元

需要根据info.fusel选择需要写回寄存器堆的wdata。写回单元的结构如图1-3所示。

如果在执行单元已经完成了写回数据的选择，则写回单元无需进行修改。



* + - 1. 写回单元

## 实验要求

1. 在实验十的基础上继续添加表1-1中列出的所有指令。
2. 通过本实验提供的所有仿真验证测试用例。
3. 通过本实验提供的所有板级验证测试用例。
4. 撰写实验报告。撰写报告时要求叙述以下内容，以及你对本实验的思考与探索。
   1. 选择需要实现的指令中的一条，按照你自己的理解，逐步介绍其数据通路设计的思路以及实现过程
   2. 更新上一实验中绘制完成的数据通路图。
5. TODO：增加更多内容

## 实验步骤

1. 实验步骤参见实验十的实验步骤部分
2. 提醒与建议
   1. 设计除法器时要注意特殊情况的处理。
   2. 建议将乘法器和除法器封装在MDU中，将MDU和ALU封装在FU中，进行统一的模块化设计。

## 思考与探索

1. 为什么乘法指令中只有MUL有对应的字指令，而别的乘法指令没有？
2. 用一条指令将128位积写入两个64位寄存器会增加硬件复杂度，能否用数据通路复用的角度解释一下原因呢？
3. 尝试实现支持乘除法指令融合的微架构设计。

1. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213。 [↑](#footnote-ref-1)
2. 指MULH、MULHU、MULHSU中的任意一种，中括号表示可选。 [↑](#footnote-ref-2)
3. 可以考虑在乘法器里保留上次的运算结果，如果源操作数和操作类型均与上一条指令一致，则可以直接使用上次的运算结果免去了重新计算的过程。 [↑](#footnote-ref-3)
4. Function Unit即功能单元。 [↑](#footnote-ref-4)