第三章学习笔记

1. 加法和减法

溢出问题：溢出发生时符号位是可能会出错的。比如两个正数相加得到负数、一个正数减去一个负数得到负结果，就有可能是溢出了。为了说明情况，有下面这张表：

操作	操作数A	操作数B	表示结果有溢出的条件
$A+B$	$\geqslant 0$	$\geqslant 0$	<0
$A+B$	<0	<0	$\geqslant 0$
$A-B$	$\geqslant 0$	<0	<0
$A-B$	<0	$\geqslant 0$	$\geqslant 0$

> 注意，有符号运算的指令才在溢出的时候产生异常，无符号运算，比如：addu，addiu，subu都是不会产生异常的。

MIPS中使用EPC(异常程序寄存器)来保存导致异常的地址，从而使mips软件通过跳转至零回到异常处。mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action. 有一些程序语言（比如C和Java）他们会忽略溢出（底层编译器使用的是addu等指令，因此不会有异常），其他的语言会通知溢出，那么就需要EPC的帮助。

> saturate： 当达到最大值以后，稳定在这个值不变，避免继续溢出

加法的操作在数字逻辑课和计算机导论课程中都已经学过了，尽量使用补码的加法，考虑溢出，这里就不再赘述。值得一提的是流水线加法器，它把一组数的加法拆分成几块分别做，每一个部分只处理新输入数据的某一段加法，并将进位(carry)传到下一部分，这样的操作可以加快加法器运算速度。流水线的优势会在第四章笔记中提到。贴一个自己在实验1里写的代码，这段代码没有考虑阻塞问题：

```
module non_Block_pipeline_adder(
  input[31:0] cin_a,
  input[31:0] cin_b,
  input c_in,
  input clk,
  output[31:0] out_c,
  output out_r //carryin
  );

  reg in1, in2, in3, in4;
  reg [7:0] sum1;
  reg [15:0] sum2;
  reg [23:0] sum3;
  reg [31:0] sum4;

  always @(posedge clk) begin
    {in1, sum1} <= cin_a[7:0] + cin_b[7:0] + c_in;
```

```
  end

  always @(posedge clk) begin
    {in2, sum2} <= {{1'b0 + cin_a[15:8]}+
            {1'b0 + cin_b[15:8]} + in1, sum1};
  end

  always @(posedge clk) begin
    {in3, sum3} <= {{1'b0 + cin_a[23:16]} +
            {1'b0 + cin_b[23:16]} + in2, sum2};
  end

  always @(posedge clk) begin
    {in4, sum4} <= {{1'b0 + cin_a[31:24]} +
            {1'b0 + cin_b[31:24]} + in3, sum3};
  end

    assign out_r = in4;
    assign out_c = sum4;

endmodule
```

2. 乘法

乘法令人恼忽，除法更甚；比例运算困扰着我，做练习令我发疯。

——佚名，《Elizabethan manuscript》，1570

$$multiplicand \times multiplier = product$$

无符号乘法算法：

1. initialize：我们在进行手算的时候可以发现，被乘数不断地乘以乘数加到积里面，这样的过程其实就是在对被乘数做sll的动作。对于一个32位乘法，被乘数要左移32位，因此我们给被乘数扩展到64位，前面用空补全。然后乘积也应当是64位，初始化为全0.
2. 测试乘数最低位=1？若是，进入到3，否则进入到4
3. 乘积加上被乘数，保存，进入4
4. 被乘数左移一位，乘数右移一位
5. 是否是第32次？是则结束，否则进入2

定点小数乘法：

$$\begin{aligned} x * y &= x(2^{-1}y_1 + 2^{-2}y_2 + \ldots + 2^{-n}y_n) \\ &= 2^{-1}(y_1x + 2^{-1}(y_2x + 2^{-1}(\ldots + 2^{-1}(y_{n-1}x + 2^{-1}(y_nx + 0))\ldots))) \end{aligned}$$

有符号乘法：

对于处理符号，最简单的方法就是先将被乘数和乘数转换为正数，并且记住原来的符号位。符号位计算只需要迭代Width - 1次，因为它有一位是拿来做符号了。对于32位，就只需要迭代31次上面的算法，最后把符号位添加在最高位就可。

3. 除法

除法的一个不同于乘法的问题在于，它的除数可能出现为0的情况，那么就是无效了。

$$dividend = quotient \times fivisor + remainder$$

除法的过程无非是不断地尝试最大能减掉多少并对应产生商。对于64位的除数和被除数，商是32位，余数是64位。我们要使用一个余数寄存器保存“被除数每次减去除数的结果”，迭代结束后的剩余就是余数。

无符号除法算法过程：

1. initialize：除数**左移**扩展到2n位，被除数**无符号扩展**到2n位。
2. 余数 - 除数，如果小于0， 进入4，否则进入3
3. 商寄存器左移，最低位设置为1，进入5
4. 余数寄存器回复原值，商寄存器左移最低位设置为0，进入5
5. 除数寄存器右移1位
6. 这是第n + 1 次迭代吗？是就结束，否进入2

建议如果是做题，先把结果算出来，再在做除法运算的过程中进行对比，减少出错。

有符号除法算法：

最简单的办法就是记住除数和被除数的符号咯，如果符号不同，商肯定就是负数了。余数也要有所改变，如果余数非零，那么它的符号要和被除数相同。

到此为止，整数型的有符号、无符号加减乘除就全部介绍完了，接下来就是一个复杂的问题：浮点数运算。

4. 浮点运算

4.1 浮点表示

尾数：位于浮点数的尾数字段，范围是0 ~ 1. 指数是浮点数的指数字段，表示小数点的位置。对于一个浮点数，这两个数值是最为重要的。除此之外，还有一个符号位。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	指数								尾数																						

1位 8位 23位

一般的浮点数如下表示：

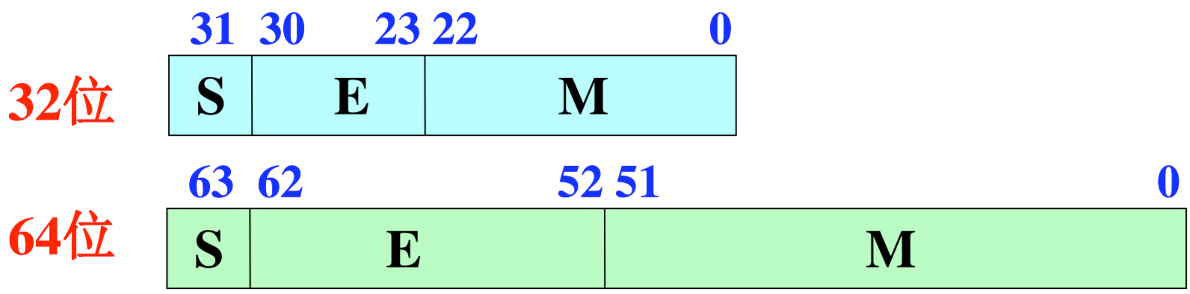
$$(-1) \times F \times R^E$$

其中F为小数域的值，E为指数域的值，R是基数，在计算机内一般是2，4，16. 浮点数也有溢出，分为上溢和下溢，很好理解，就分别是正的指数太大和负的指数太大而导致放不下。

尾数：用定点小数表示，给出有效数字的位数，决定了浮点数的表示精度

阶码：用定点整数形式表示，指明小数点在数据中的位置，决定了浮点数的表示范围

4.2 IEEE 754 （重要）



规格化处理：在计算机内，其纯小数部分被称为浮点数的尾数，对非 0 值的浮点数，要求尾数的绝对值必须 >= ½，即尾数域的最高有效位应为1。不过考虑到正数和负数之间的差别，其实是尾数的最高位与符号位**相反**。

隐藏位技术：既然非 0 值浮点数的尾数数值最高位必定为 1，则在保存浮点数到内存前，通过尾数左移，强行把该位去掉,用同样多的尾数位就能多存一位二进制数，有利于提高数据表示精度，称这种处理方案使用了隐藏位技术。在取回这样的浮点数到运算器执行运算时，必须先恢复该隐藏位

IEEE 754 规定单精度的偏阶为127， 双精度为1023，这种技术方法需要从带偏阶的指数中减去偏阶才能得到真值。那么对于一个用IEEE 754表示的单精度数，对于S，E，M，对其的真值计算如下：

$$val = (-1)^S \times (1.M) \times 2^{E-127}$$

例题参考ppt，我这里就不另外贴了。

IEEE 754 表示数的范围：

格式	最小值	最大值
单精度	E=1, M=0, $1.0 \times 2^{1-127} = 2^{-126}$	E=254, f=.1111 …, $1.111\ldots1 \times 2^{254-127} = 2^{127} \times (2-2^{-23})$
双精度	E=1, M=0, $1.0 \times 2^{1-1023} = 2^{-1022}$	E=2046, f=.1111 …, $1.111\ldots1 \times 2^{2046-1023} = 2^{1023} \times (2-2^{-52})$

有一些规定：

1. E = 0 且 M = 0，则真值为0
2. E = 0 且 M != 0，为*非规格化数*，真值 = $(-1) * 0.M * 2^{-126}$
3. 1<= E <= 254，按照前面的标准计算真值
4. E = 255 且 M != 0, 真值为“NaN”
5. E = 255 且 M = 0, 真值为由符号位决定的无穷值。

从上面的表看，可能感觉浪费了两位没表示，但其实是这两部分是用来处理数的一些异常值的，用他们来规定边界，才能让程序更好地执行。

4.3 浮点加减法

对于两个数， $x = 2^{Ex} \cdot Mx$， $y = 2^{Ey} \cdot My$， 两个的加减法结果为：

$$x \pm y = (Mx \cdot 2^{Ex-Ey} \pm My) \cdot Ey \quad (Ex <= Ey)$$

这个计算过程最重要的就是要把指数较小的数进行右移，直到他的指数和指数较大的相匹配。但是，远远没有这么简单，我们之前介绍了ieee 754表示的数的范围，为什么会这样？因为加减乘除是有可能产生溢出的！下面我们来完整地讨论加减法过程。

加减法的基本流程如下：

> (1) 0 操作数的检查；
>
> (2) 比较阶码大小并完成对阶；对阶？就是让两个数的小数点位对齐，对齐的原则是**阶码小的数向阶码大的数对齐**
>
> (3) 尾数进行加或减运算；
>
> (4) 结果规格化。规格化就是为了避免尾数过大或者过小。如果补码表示的情况下出现符号位和尾数最高位相等，说明尾数小了， 需要进行左规划；如果计算的结果双符号位变成了*01*或者*10*，说明尾数太大，溢出位破坏了规格化，此时向右边规划。规格化什么时候结束？符号位和尾数最高位相异。
>
> (5) 舍入处理。① **“0舍1入”法**
>
> 如果右移时被丢掉数位的最高位为0则舍去，反之则将尾数的末位加1
>
> **② “恒置1“法**
>
> 即只要数位被移掉，就在尾数的末位恒置“1”。从概率上说，丢掉的0和1各为1/2
>
> (6)溢出处理。
>
> 当尾数之和(差)出现01．××…× 或10．××…×时，并不表示溢出，只有将此数右规后，再根据阶码来判断浮点运算结果是否溢出。

4.4 浮点数乘除法

对于两个数， $x = 2^{Ex} \cdot Mx$， $y = 2^{Ey} \cdot My$，有$x \times y = 2^{Ex+Ey}(MxMy)$， 以及 $x \div y = 2^{Ex-Ey}(Mx \div My)$。总的来说，计算大概也就4个步骤：

> (1) 0 操作数检查；
> (2) 阶码加/减操作；
> (3) 尾数乘/除操作；
> (4) 结果规格化及舍入处理。

阶码运算结果溢出处理： 使用双符号位的阶码加法器, 并规定移码的第二个符号位, 即最高符号位恒用 0 参加加减运算, 则溢出条件是结果的最高符号位为1：

• 当低位符号位为 0时,（10） 表明结果上溢,
• 当低位符号位为1时, （11） 表明结果下溢。
• 当最高符号位为0时, 表明没有溢出:
低位符号位为1, （01） 表明结果为正;
为0, （00） 表明结果为负。