RV32F 和 RV32D:单精度和双精度浮点数

只有当没有任何东西可以去除，而不是没有东西可以添加时，我们才最终达到了完美。

5．1 介绍
尽管 RV32F 和 RV32D 是分开的，单独的可选指令集扩展，他们通常是包括在一起的。为简洁起见，我们在一章中介绍了几乎所有的单精度和双精度 (32 位和 64 位) 浮点指令。图 5.1 是一个 RV32F 和 RV32D 扩展指令集的图形表示。图 5.2 列出 RV32F 的操作码，图 5.3 列出了 RV32D 的操作码。和几乎所有其他现代 ISA 一样，RISC-V 服从 IEEE 754-2008 浮点标准[IEEE 标准委员会 2008]

5.2 浮点寄存器
RV32F 和 RV32D 使用 32 个独立的 f 寄存器而不是 x 寄存器。使用两组寄存器的主要原因是：处理器在不增加 RISC-V 指令格式中寄存器描述符所占空间的情况下使用两组寄存器来将寄存器容量和带宽是乘 2，这可以提高处理器性能。使用两组寄存器对 RISC-V 指令集的主要影响是，必须要添加新的指令来加载和存储数据 f 寄存器，还需要添加新指令用于在 x 和 f 寄存器之间传递数据。图 5.4 列出了 RV32D 和 RV32F 寄存器及对应的由 RISC-V ABI 确定的寄存器名称。

如果处理器同时支持 RV32F 和 RV32D 扩展，则单精度数据仅使用 f 寄存器中的低 32 位。与 RV32I 中的 x0 不同，寄存器 f0 不是硬连线到常量 0，而是和所有其他 31 个 f 寄存器一样，是一个可变寄存器。

IEEE 754-2008 标准提供了几种浮点运算舍入的方法，这有助于确定误差范围和编写数值库。最准确且最常见的舍入模式是舍入到最近的偶数（RNE）。舍入模式可以通过浮点控制和状态寄存器 fcsr 进行设置。图 5.5 显示了 fcsr 并列出了舍入选项。它还包含标准所需的累积异常标志。

有什么不同？ ARM-32 和 MIPS-32 都有 32 个单精度浮点寄存器但都只有 16 个双精度寄存器。它们都将两个单精度寄存器映射到双精度寄存器的左右两半。 x86-32 浮点数算术没有任何寄存器，而是使用堆栈代替。堆栈条目是 80 位宽度提高精度，因此浮点数负载将 32 位或 64 位操作数转换为 80 位，对于存储指令，反之亦然。 x86-32 的一个后续版本增加了 8 个传统的 64 位浮点寄存器以及相关的操作指令。与 RV32FD 和 MIPS-32 不同，ARM-32 和 x86-32 忽视了在浮点和整数寄存器之间直接移动数据的指令。唯一的解决方案是先将浮点寄存器的内容存储在内存中，然后将其从内存加载到整数寄存器，反之亦然。

详细说明：RV32FD 允许逐条指令设置舍入模式。
这被称为静态舍入，当你只需要更改一条指令的舍入模式时，它可以帮助提高性能。默认设置是在 fcsr 中使用动态舍入模式。静态舍入所选择的模式是作为指令可选的最后一个参数，

如 fadd.s ft0，ft1，ft2，rtz，将向零舍入，无论 fcsr 如何。图 5.5 的标题列出了不同舍入模式的名称。

5.3 浮点加载，存储和算术指令

对于 RV32F 和 RV32D，RISC-V 有两条加载指令（flw，fld）和两条存储指令（fsw，fsd）。他们和 lw 和 sw 拥有相同的寻址模式和指令格式。添加到标准算术运算中的指令有：（fadd.s，fadd.d，fsub.s，fsub.d，fmul.s，fmul.d，fdiv.s，fdiv.d），RV32F 和 RV32D 还包括平方根（fsqrt.s，fsqrt.d）指令。它们也有最小值和最大值指令（fmin.s，fmin.d，fmax.s，fmax.d），这些指令在不使用分支指令进行比较的情况下，将一对源操作数中的较小值或较大值写入目的寄存器。

图 5.2：RV32F 操作码表包含了指令布局，操作码，格式类型和名称。这张表与下一张表在编码上的主要区别是：对于这张表，前两个指令第 12 位是 0，并且对于其余指令，第 25 位为 0，而在下一张表中，RV32D 中的这两个位均为 1。 ([Waterman and Asanovic 2017]是这个图表的基础。

图 5.3：RV32D 操作码表包含了指令布局，操作码，格式类型和名称。这两个图中的一些指令并不仅仅是数据宽度不同。只有这张表有 fcvt.s.d 和 fcvt.d.s 指令，而只有另一张表有 fmv.x.w 和 fmv.w.x.指令。（[Waterman and Asanovic 2017]的表 19.2 是这张表的基础。）

图 5.4：RV32F 和 RV32D 的浮点寄存器。单精度寄存器占用了 32 个双精度寄存器中最右边的一半。第 3 章解释了 RISC-V 对于浮点寄存器的调用约定，阐述了 FP 参数寄存器（fa0-fa7），FP 保存寄存器（fs0-fs11）和 FP 临时寄存器（ft0-ft11）背后的基本原理。（[Waterman and Asanovic 2017]的表 20.1 是这个表的基础）

图 5.5：浮点控制和状态寄存器。它包含舍入模式和异常标志。

？？？？？

舍入模式是圆形到最近的，与偶数（rte，000 in frm）相关;向零舍入（rtz，001）;回合向下，朝-1（rdn，010）;向上，向+1（rup，011）;并且绕到最近，与最大值相关联幅度（rmm，100）。五个应计异常标志表示出现的异常情况自上次由软件重置字段以来的任何浮点算术指令：NV 是无效操作;DZ 除以零; OF 溢出; UF 是下溢;和 NX 是不精确的。 ([Waterman and Asanovic 2017]是这个数字的基础。）

许多浮点算法（例如矩阵乘法）在执行完乘法运算后会立即执行一条加法或减法指令。因此，RISC-V 提供了指令用于先将两个操作数相乘然后将乘积加上（fmadd.s，fmadd.d）或减去（fmsub.s，fmsub.d）第三个操作数，最后再将结果写入目的寄存器。它还有在加上或减去第三个操作数之前对乘积取反的版本：fnmadd.s，fnmadd.d，fnmsub.s，fnmsub.d。这些融合的乘法 - 加法指令比单独的使用乘法及加法指令更准确，也更快，因为它们只（在加法之后）舍入过一次，而单独的乘法及加法指令则舍入了两次（先是在乘法之后，然后在加法之后）。这些指令需要一条新指令格式指定第 4 个寄存器，称为 R4。图 5.2 和 5.3 显示了 R4 格式，它是 R 格式的一个变种。

RV32F 和 RV32D 没有提供浮点分支指令，而是提供了浮点比较指令，这些根据两个浮点的比较结果将一个整数寄存器设置为 1 或 0：feq.s，feq.d，flt.s，flt.d，fle.s，fle.d。这些指令允许整数分支指令根据浮点数比较指令设置的条件进行分支跳转。例如，这段代码在 f1 <f2 时，则分支跳转到 Exit：

```
flt x5，f1，f2　　# 如果 f1 < f2，则 x5 = 1;否则 x5 = 0
bne x5，x0，Exit　# 如果 x5！= 0，则跳转到退出
```

5.4 浮点转换和搬运

RV32F 和 RV32D 支持在在 32 位有符号整数，32 位无符号整数，32 位浮点和 64 位之间浮点进行所有组合的转换（只要这个转换是有用，有意义的）。图 5.6 按源数据类型以及转换后的目的数据类型，罗列了这 10 条指令。

RV32F 还提供了将数据从 f 寄存器（fmv.x.w）移动到 x 寄存器的指令，以及反方向移动数据的指令（fmv.w.x）。

5.5 其他浮点指令

RV32F 和 RV32D 提供了不寻常的指令，有助于编写数学库以及提供有用的伪指令。(IEEE 754 浮点标准需要一种复制并且操作符号并对浮点数据进行分类的方式，这启发我们添加了这些指令。)

第一个是符号注入指令，它从第一个源操作数复制了除符号位之外的所有内容。符号位的取值取决于具体是什么指令：

1.浮点符号注入（fsgnj.s，fsgnj.d）：结果的符号位是 rs2 的符号位。
2.浮点符号取反注入（fsgnjn.s，fsgnjn.d）：结果的符号位与 rs2 的符号位相反
3.浮点符号异或注入（fsgnjx.s，fsgnjx.d）：结果符号位是 rs1 和 rs2 的符号位异或的结果

除了有助于数学库中的符号操作，基于符号注入指令我们还提供了三种流行的浮点伪指令（参见第 37 页的图 3.4）：

1.复制浮点寄存器：

```
fmv.s rd，rs 事实上是 fsgnj.s rd，rs，rs
fmv.d rd，rs 事实上是 sgnj.d rd，rs，rs。
```

2.否定：

```
fneg.s rd，rs 映射到 fsgnjn.s rd，rs，rs
fneg.d rd，rs 映射到 fsgnjn.d rd，rs，rs。
```

3.绝对值（因为 0⊕0= 0 且 1⊕1= 0）：

```
fabs.s rd，rs 变成了 fsgnjx.s rd，rs，rs
fabs.d rd，rs 变成了 sgnjx.d rd，rs，rs。
```

第二个不常见的浮点指令是 classify 分类指令（fclass.s，fclass.d）。分类指令对数学库也很有帮助。他们测试一个源操作数来看源操作数满足下列 10 个浮点数属性中的哪些属性（参见下表），然后将测试结果的掩码写入目的整数寄存器的低 10 位。十位中仅有一位被设置为 1，其余为都设置为 0。

```
x [rd]位　　含义
0　　f [rs1]为负无穷大。
```

1 f [rs1]是负正常数。
2 f [rs1]是负的次正规数。
3 f [rs1]是-0。
4 f [rs1]是+0。
5 f [rs1]是正次正规数。
6 f [rs1]是正的正常数。
7 f [rs1]为+1。
8 f [rs1]是信号 NaN。
9 f [rs1]是一个安静的 NaN。
这里怎么翻译？正常数？还是规范数？要复习一下数据表示了，，，

```
void daxpy(size_t n, double a, const double x[], double y[])
{
for (size_t i = 0; i < n; i++) {
y[i] = a*x[i] + y[i];
}
}
```

图 5.7:用 C 编写的 浮点运算密集型的 DAXPY 程序

指令集	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32FD	RV32FD+RV32C
全部指令数	10	10	12	12	16	11	11
每循环指令数	6	6	7	7	6	7	7
字节数	40	28	48	32	50	44	28

图 5.8：DAXPY 在四个 ISA 上生成的指令数和代码大小。它列出了每个循环的指令数量以及指令总数。第 7 章介绍 ARM Thumb-2，microMIPS 和 RV32C 指令集。

5.6 使用 DAXPY 程序比较 RV32FD，ARM-32，MIPS-32 和 x86-32

我们现在将使用 DAXPY 作为我们的浮点基准对不同 ISA 进行比较（图 5.7）。它以双精度计算 Y = a×X + Y，其中 X 和 Y 是矢量，a 是标量。图 5.8 总结了 DAXPY 在四个不同的 ISA 下对应的指令数和字节数。他们的代码如图 5.9 至 5.12 所示。
DAXPY 这个名字来自公式本身：以双精度计算 A 乘上 X 加 Y.此公式的单精度版本被称做 SAXPY。
与第 2 章中的插入排序一样，尽管 RISC-V 指令集强调简单性，RISC-V 版本的不管是指令数量还是代码大小，都接近或者优于其他 ISA。在此示例中，RISC-V 的比较和执行分支指令和 ARM-32 和 x86-32 中更复杂的寻址模式，以及入栈、退栈指令节省了差不多数量的指令。

5.7 结束语

少即是多。

IEEE 754-2008 浮点标准[IEEE 标准委员会 2008]定义了浮点数据类型，计算精度和所需操作。它的广泛流行大大降低了移植浮点程序的难度，这也意味着不同 ISA 中的浮点数部分可能比其他章节中描述的其他部分的指令更一致。

详细说明：16 位，128 位和十进制浮点运算

修订后的 IEEE 浮点标准（IEEE 754-2008）除了单精度和双精度之外，还描述了几种新的浮点数格式，它们称为 binary32 和 binary64。不出意料，修订后，新增了四倍精度，名为 binary128。 RISC-V 暂时计划用 RV32Q 扩展来支持新的浮点数格式（见第 11 章）。该标准还为二进制数据交换提供了两种新的数据尺寸，程序员可以会将这些浮点数以特定宽度存储在内存或存储中，但是或许不能以这种宽度进行计算计算。它们分别是半精度（binary16）和八元精度（binary256）。尽管标准对这两种新宽度的存储使用定义的，但 GPU 确实以半精度计算并将它们保存在内存中。 RISC-V 的计划在向量指令（第 8 章中的 RV32V）中包含半精度计算，但是前提是处理器如果支持向量半精度指令，则也必须支持半精度标量指令。令人惊讶的是，修订后标准还添加了十进制浮点数，新增的三种十进制格式分别是 decimal32，decimal64 和 decimal128。RISC-V 预留 RV32L 指令集扩展用于支持它（见第 11 章）。

5.8 了解更多

RV32FD (循环体 7 条指令；总计 11 条指令 /44 字节大小；使用 RVC 指令集，是 28 字节)

a0 是 n, a1 是指向 x[0]的指针, a2 是指向 y[0]的指针, fa0 是 a

0: 02050463 beqz a0,28 # 如果 n== 0, 跳转到 Exit

4: 00351513 slli a0,a0,0x3 # a0 = n*8

8: 00a60533 add	a0,a2,a0	# a0 = address of x[n] (last element)
Loop: c: 0005b787 fld	fa5,0(a1)	# fa5 = x[]
10: 00063707 fld	fa4,0(a2)	# fa4 = y[]
14: 00860613 addi	a2,a2,8	# a2++ (increment pointer to y)
18: 00858593 addi	a1,a1,8	# a1++ (increment pointer to x)

1c: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]

20: fef63c27 fsd fa5,-8(a2) # y[i] = a*x[i] + y[i]

24: fea614e3 bne a2,a0,c # if i != n, jump to Loop

Exit:

28: 00008067 ret # return

图 5.9：图 5.7 中 DAXPY 的 RV32D 代码。十六进制的地址位于机器的左侧，接下来是十六进制的语言代码，然后是汇编语言指令，最后是注释。比较和分支指令避免了 ARM-32 和 X86-32 代码中的两条比较指令。

ARM-32 (6 insns in loop; 10 insns/40 bytes total; 28 bytes Thumb-2)

r0 is n, d0 is a, r1 is pointer to x[0], r2 is pointer to y[0]

0: e3500000 cmp r0, #0 # compare n to 0

4: 0a000006 beq 24 <daxpy+0x24> # if n == 0, jump to Exit

8: e0820180 add	r0, r2, r0, lsl #3 # r0 = address of x[n] (last element)	
Loop: c: ecb16b02 vldmia	r1!,{d6}	# d6 = x[i], increment pointer to x
10: ed927b00 vldr	d7,[r2]	# d7 = y[i]
14: ee067b00 vmla.f64 d7, d6, d0	# d7 = a*x[i] + y[i]	
18: eca27b02 vstmia	r2!, {d7}	# y[i] = a*x[i] + y[i], incr. ptr to y
1c: e1520000 cmp	r2, r0	# i vs. n
20: 1afffff9 bne	c <daxpy+0xc>	# if i != n, jump to Loop
Exit: 24: e12fff1e bx	lr	# return

图 5.10：图 5.7 中 DAXPY 的 ARM-32 代码。 与 RISC-V 相比，ARM-32 的自动增量寻址模式可以节省两条指令。与插入排序不同，DAXPY 在 ARM-32 上不需要压栈和出栈寄存器。

MIPS-32 (7 insns in loop; 12 insns/48 bytes total; 32 bytes microMIPS)

a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], f12 is a

0: 10800009 beqz a0,28 <daxpy+0x28> # if n == 0, jump to Exit

4: 000420c0 sll a0,a0,0x3 # a0 = n*8 (filled branch delay slot)

8: 00c42021 addu	a0,a2,a0	# a0 = address of x[n] (last element)
Loop: c: 24c60008 addiu	a2,a2,8	# a2++ (increment pointer to y)
10: d4a00000 ldc1	$f0,0(a1)	# f0 = x[i]
14: 24a50008 addiu	a1,a1,8	# a1++ (increment pointer to x)
18: d4c2fff8 ldc1	$f2,-8(a2)	# f2 = y[i]
1c: 4c406021 madd.d $f0,$f2,$f12,$f0	# f0 = a*x[i] + y[i]	
20: 14c4fffa bne	a2,a0,c <daxpy+0xc> # if i != n, jump to Loop	
24: f4c0fff8 sdc1	$f0,-8(a2)	# y[i] = a*x[i] + y[i] (filled delay slot)
Exit: 28: 03e00008 jr	ra	# return
2c: 00000000 nop	# (unfilled branch delay slot)	

图 5.11：图 5.7 中 DAXPY 的 MIPS-32 代码。三个分支延迟槽中的两个填充了有用的指令。检查两个寄存器之间是否相等的指令避免了 ARM-32 和 x86-32 中的两条比较指令。与整数加载不同，浮点加载没有延迟槽。

```
# x86-32 (6 insns in loop; 16 insns/50 bytes total)
# eax is i, n is in memory at esp+0x8, a is in memory at esp+0xc
# pointer to x[0] is in memory at esp+0x14
# pointer to y[0] is in memory at esp+0x18
0: 53 push ebx # save ebx
1: 8b 4c 24 08 mov ecx,[esp+0x8] # ecx has copy of n
5: c5 fb 10 4c 24 0c vmovsd xmm1,[esp+0xc] # xmm1 has a copy of a
b: 8b 5c 24 14 mov ebx,[esp+0x14] # ebx points to x[0]
f: 8b 54 24 18 mov edx,[esp+0x18] # edx points to y[0]
13: 85 c9 test ecx,ecx # compare n to 0
15: 74 19 je 30 <daxpy+0x30> # if n==0, jump to Exit
17: 31 c0 xor eax,eax # i = 0 (since x^x==0)
Loop:
19: c5 fb 10 04 c3 vmovsd xmm0,[ebx+eax*8] # xmm0 = x[i]
1e: c4 e2 f1 a9 04 c2 vfmadd213sd xmm0,xmm1,[edx+eax*8] # xmm0 = a*x[i] + y[i]
24: c5 fb 11 04 c2 vmovsd xmm0,xmm1,[edx+eax*8] # y[i] = a*x[i] + y[i]
29: 83 c0 01 add eax,0x1 # i++
2c: 39 c1 cmp ecx,eax # compare i vs n
2e: 75 e9 jne 19 <daxpy+0x19> # if i!=n, jump to Loop
Exit:
30: 5b pop ebx # restore ebx
31: c3 ret # retu
```

图 5.12：图 5.7 中 DAXPY 的 x86-32 代码。在这个例子中，x86-32 缺少寄存器的劣势在这里表现得很明显——有四个变量被分配到了内存，而在其他 ISA 中，这些变量是被存放在寄存器中的。它展示了 x86-32 中，如何将寄存器与零比较（测试 ecx，ecx）以及如何将一个寄存器清零（xor EAX，EAX）。