CUDA SASS汇编器实现笔记（3）-指令编码的特殊处理

https://zhuanlan.zhihu.com/p/314410214

None Fri Jun, 04 04:31

上期已经将SASS指令自动编码的通用方法大致介绍了一遍：

cloudcore：CUDA SASS汇编器实现笔记（2）-自动指令编码zhuanlan.zhihu.com

然而，通用方法并不能覆盖所有的指令。毕竟作为一个实用的指令集，总是会有一些需要变通的地方。而且，规则是NV定的，我们的目的是尽量与nvdisasm的汇编形式上的逻辑保持一致。最关键的是指令编码容错空间其实很小，如果一个出了问题很可能整个程序都无法正常运行。所以，为了保证实现的完备性和可靠性，我们还是需要加入一些特殊处理，期望能完整的复现所有的编码情况。

先简略回顾一下通用方法，所有指令按Opcode和operand的类型分类。每类指令的编码值都化为值value与权weight的内积。其中value与上下文无关，通过汇编文本即可求出；而weight则与上下文和位置有关，是待求值。通过收集许多已知指令编码的不同的value序列，构成一个线性代数方程组，求解出相应权重。当需要编码新指令时，用其value与解出的weight求内积即可得到编码值。例如：

```
/*1190*/    @P0 FADD.FTZ R13, -R14, -RZ ;    /* 0x800000ff0e0d0221 */
                                             /* 0x000fc80000010100 */
```

可以化为：

```
c(*) = c('@P0') + c('FADD') + c('FTZ') + c('0_R13') + c('1_Neg') + c('1_R14') + c('2_Neg') + c('2_R255').
     = [0] * W('@P#') + [1] * W('FADD') + [1] * W['FTZ'] + [13]*W['0_R#'] + [1] * W['1_Neg'] + [14] * W['1_R#']
       + [1] * W['2_Neg'] + [255] * W['2_R#']
     = [0, 1, 1, 13, 1, 14, 1, 255] * [w('@P#'), w('FADD'), w('FTZ'), w('0_R#'), w('1_Neg'), w('1_R#'), w('2_Neg'), w('2_R#')]^T
     = v * w^T
```

前一期已经说过，通用方法可以处理所有满足线性形式（`c=v*w^T`）的指令，而且即使在输入不充分导致系数矩阵亏秩时仍然可以编码部分指令。那需要特殊处理主要分两种，一是指令的某个operand不知道如何确定value。二是指令编码根本不是value的线性组合，也就是线性相关关系被打破的情况。

这期介绍的这些需要特殊处理的情况，对于其他编码算法也是需要特殊处理的。如果你想自己用别的方式来做编码，这些也会有些参考价值。

这里为了阐述方便，主要用Turing的指令编码做介绍，当与Maxwell/Pascal有显著差别时，一般会单独提出来。

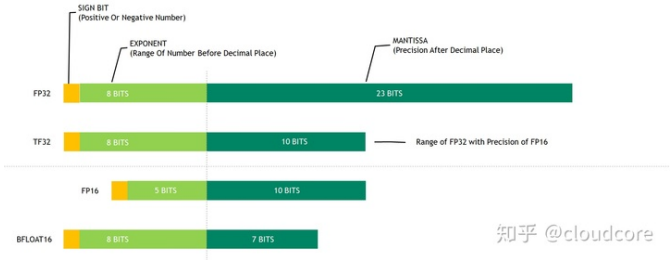
值不确定的情况

首先先说value未知的问题。一般的指导思想就是根据类型和opcode定一个确定的映射规则。operand的值的分类情况上期已经讲过，多数情况可以简单求值。不能显式求值的，其实主要就两种情况：一种是浮点数，一种是string形式的label。还有一种是比较少见的 `DEPBAR` 指令的Scoreboard的集合情况。

浮点立即数

浮点数的二进制化主要问题在于如何决定其所用的格式，以及指令中使用的bit位数。

首先，是确定编码用的**浮点格式**。即使是简单的1，作为F32、F16、F64的二进制表示都是不一样的，因为三者的指数位的位数是不一样的，需要区别编码（具体大家可以去参考IEEE 754，最新版是2019）。这里借NV的一张图来说明一下（原图来自Accelerating TensorFlow on NVIDIA A100 GPUs）。这里没写double，它的指数有11bit，尾数52bit。



A100支持的浮点格式（省略了double）

不过，浮点立即数一般只会出现在ALU指令中，而SASS的大多数浮点ALU指令的opcode的首字母都会自我表明身份，比如 `DFMA, DADD, DMUL` 是**D**ouble（F64），`FFMA, FADD, FMUL` 是**F**loat（F32），`HFMA2, HADD2, HMUL2` 是**H**alf（F16）。其他如 `I2F` 、 `F2I` 、 `F2F` 之类的理论上也会可能出现浮点立即数。但据我现在的观察，这些指令都只有一个源操作数，绝大多数情况下编译器能搞定这个转换，所以一般不需要这种方式。 `MOV` 理论上也可能支持浮点立即数，但是在操作数直接支持的情况下，只有需要至少两个立即数的时候才需要MOV。另外，MOV只赋值不做运算，所以直接当成整形应该也可以，所以汇编里还没怎么见到过这种形式。因此，对绝大多数涉及浮点的ALU指令，可以通过首字母

一个例外是 `MUFU` ，比如这种：

```
MUFU.RCP64H R3, 29.999998092651367188 ;
```

这也挺奇怪的（这个是官方库里发现的……）。让编译器把结果算出来直接赋给`R3`不行吗？说不定还真不行！因为`MUFU`都是近似计算，编译器真不一定有能力精确还原硬件的计算结果。有时候为了与原语义精确保持一致，编译器对有些可能改变结果的优化也许是有顾忌的。特别是不满足结合律的浮点运算。比如交换次序：`(a+b)-a != a+(b-a)`，fma融合：`fma(a,b,c)!= a*b+c`。这些操作都可能导致结果发生变化。这个涉及到很多浮点运算中微妙的问题。有兴趣的可以参考754的原文和David Goldberg的《What every computer scientist should know about floating-point arithmetic》。

【注】：话是这么说……但是我直接用`__sinf(1.0f)`之类编译器确实会用精确值算出来。所以即使`v=1.0f`，也可能有`__sinf(v) != __sinf(1.0f)`。没仔细研究过编译器现在是怎么控制这个。只记得默认有`--fmad=true`，近似除法和`FTZ`也有开关，但是这个没看到有开关。

我也不太确定这里`MUFU`到底是为啥这么用（我用CUDA C试了很多次，好像没法触发这个指令……），但既然这个格式也能从modifier(`RCP64H`)里看出来，我只需要对opcode为`MUFU`的指令再加个特殊处理即可。另外，从这个指令可以看出64bit的`MUFU`只用了一个GPR输入输出，精度不会超过32bit。因为如果用两个，GPR的对齐要求`R#`需要为偶数，但这里用的`R3`，显然不是对齐的。说明它只是用单个GPR参与的近似计算。当然，尽管是32bit，它应该是64bit浮点数的前32bit，并不是float32的格式。

其次，要决定浮点数的**编码位数**。对于完整的浮点格式，F64、F32、F16显然各自是64bit、32bit、16bit。但volta前的一条指令总共就64bit，显然不能都用来编码立即数，所以肯定会有所截断。一般说来，这种截断都是在尾数significand部分（以前都叫mantissa，但现在754的标准叫法是significand。中文一般叫尾数，就是二进制表示的尾部），但是保持指数位和符号位不变。因此，得到浮点立即数的完整二进制表示后，做移位就可以得到所需的编码部分。

就我现在看到的情况来看，Turing的F64、F32立即数都是32bit，F16是16bit。Maxwell/Pascal比较烦，一般的F64、F32看起来是20bit，但比较坑的是符号位好像会被剥离出来，而把对应operand取负的modifier的bit置上。【注】：其实我也没看到原来那个符号bit做别的用处，一般都是空着，两者应该是等效的。这个其实本来我也不用管，把负号剥离出来做独立的modifier，它自己会去找位置。但如果两种方式同时存在，就会有同一个汇编文本对应两种编码的问题，那个方程就会有问题。

【更正】：开始没理清楚，仔细核对了一下SM50中float立即数的编码，发现符号位确实被占用了（FFMA的第三个src操作数）。至少FFMA是这样，不过这对我的处理影响不大，反正就是把负号作为独立自由度就可以了。

```
0x338012bf00072225  FFMA R37, R34, -0.5, R37 ;
0011 0011 1000 0000 0001 0010 1011 1111 0000 0000 0000 0111 0010 0010 0010 0101
3    3    8    0    1    2    b    f    0    0    0    7    2    2    2    5

0x3281103f00072222  FFMA R34, R34, 0.5.NEG, R32 ;
0011 0010 1000 0001 0001 0000 0011 1111 0000 0000 0000 0111 0010 0010 0010 0010
3    2    8    1    1    0    3    f    0    0    0    7    2    2    2    2

0x328011bf00072a23  FFMA R35, R42, 0.5, R35 ;
0011 0010 1000 0000 0001 0001 1011 1111 0000 0000 0000 0111 0010 1010 0010 0011
3    2    8    0    1    1    b    f    0    0    0    7    2    a    2    3
```

Maxwell/Pascal有些指令（opcode中有32I的，如 `FADD32I` ）支持32bit的立即数。更复杂的我还没理清楚，感觉这里面还是有不少坑。Turing把编码弄长就没这么多事了，带32I的那些指令似乎还在，但是没差别了，将来说不定就去掉了。

由于指数位不受影响，significand的bit截断对浮点立即数的动态范围影响不大，但会限制了它的表示精度。这也意味着有些需要高精度输入的场合，浮点立即数不一定能帮上忙。比如对于很多数学函数的精确求值，都会采用多个FMA来进行多项式求值（当年我们老师特地强调这是秦九韶法~外国人叫霍纳算法），例如下面这个3次多项式：

```
p(x) = a x^3 + b x^2 + c x + d = fma(x, fma(x, fma(x, a, b), c), d).
```

其中的系数往往精度要求很高，几乎每个bit都要用上。对于Turing来说，float立即数的位数是满的32位，因而这些系数可以用立即数表示。但double也只有32bit，那就只能把系数放在constant memory里。对于Pascal和Maxwell，float的位数也不满，所以这个也只好都放在constant memory里。当然，用32bit立即数的 `MOV` 也行，但是会多消耗一些指令。

NV最近的架构加入了一些新的浮点形式，比如BF16、TF32之类，暂时还没看到这种格式立即数的需求。如果需要，感觉可以用F32的截断来做，因为它们指数位和F32是一样多，只是尾数长度不一样。

浮点还有几类特殊的数，比如 `NAN, +INF/-INF` 之类。 `INF` 的支持没看到有什么问题，格式和位数对就行。但是 `NAN` 却有毛病。暂时我只看到 `QNAN` 的情况，如下面这种（暂时只看到出现在 `FSEL` 中，其他应该确实也不需要，因为多数都能直接算出来）：

```
FSEL R5, R5, +QNAN , P0 ;
```

这里的一个大麻烦是不论是**QNAN**还是**SNAN**，它都不是单个数，而是一**组**数。按照规范，如果浮点指数位是全1，尾数第一位为1就是 `QNAN` ，尾数第一位是0后面非0则为 `SNAN` 。如果尾数所有位都为0，那就是 `INF` 。因此， `SNAN` 和 `QNAN` 的尾数除了第一位确定，其余位是不确定的（ `SNAN` 要求不全为0，否则就变成 `INF` 了）。因此，光给出 `QNAN` 或 `SNAN` ，并不能确定它具体的二进制表示。这其中也有一些骚操作，比如以前就有把数组初始化为一堆 `NAN` ，后面的尾数

bit初始化为数组索引。正常来说这部分会在程序中被初始化为normal值。而如果没有正确初始化的话，由于浮点运算中 `NAN` 会不断传播，最后报上错来看 `NAN` 的尾数bit就可以看出具体哪个索引位置没初始化对。这里我没看出这个逻辑来，也许是有别的编码方式。但是，由于这个信息不在汇编文本里，就没法恢复。所以我这里增加了另一种显式的浮点立即数的指定方法，如 `0F3f800000` （这也是PTX用的方式）。这有另一个好处是减少进制转换可能带来的舍入误差。不过，如果要复现原指令的含义，我们就只好在反汇编的时候就预先做这个处理，否则反汇编成 `QNAN` 后这个信息就丢掉了。

Label的处理

很多指令会出现一些字符串标签形式的的操作数:

```
S2R R3, SR_TID.X ;
S2R R23, SR_TID.Y ;
S2R R11, SR_CTAID.X ;
S2R R2, SR_CTAID.Y ;
R2P PR, R191 ;
P2R.B3 R142, PR, R142, 0x78 ;
CS2R R4, SRZ ;
S2R R78, SR_GEMASK ;
S2R R0, SR_LANEID ;
TEX.SCR.LL RZ, R6, R4, R11, 0x0, 0x5e, 2D, 0x1 ;
```

就我现在的观察来看，多数标签只是为了表示操作的方式或对象，其实放在操作数里和放在opcode的modifier里差别不是很大。比如 `S2R R3, SR_TID.X` 就可以认为是 `S2R.SR_TID.X R3` ，这个指令 `S2R.SR_TID.X` 的目的就是把threadIdx.x载入到对应GPR中。多数情况下，把label作为opcode或operand的modifier差别不大。反正只要有足够的输入，这些都能被编码出来。不过，我这里偷了懒，直接把label塞在了Key里，也就是 `S2R` 和 `SR_TID.X` 一起来决定整体的值。像 `SR_TID` ， `SR_CTAID` 这种对象，有 `X, Y, Z` 三个分量，我这里也没拆分。因为我也不保证 `SR_TID` 、 `SR_CTAID` 和 `X,Y,Z` 之间一定是独立正交的关系，反正作为整体一定不会犯错。其实这个操作空间很大，应该说是只要自由度给够，怎么弄都行，反正我也不关心底层逻辑，它也不影响编码结果。

DEPBAR的Scoreboard集合操作数

`DEPBAR` 指令支持等待一串Scoreboard的操作。如:

```
DEPBAR.LE SB0, 0x0, {2,1} ;
DEPBAR.LE SB1, 0x0, {4,3,2} ;
```

这也许是个legacy的指令。因为Maxwell之后，control codes里已经可以直接设置同时等待多个Scoreboard为0，多数情况下不用专门写这条指令了。 `DEPBAR` 的作用应该主要还是等某个Scoreboard的计数不为0的情况（比如 `DEPBAR.LE SB5, 0x6 ;` 表示stall直至第5个Scoreboard的

值降到6或以下），但这时一般就不太需要等待多个Scoreboard。话虽这么说，Maxwell的code里偶尔还能见到类似 `DEPBAR {0} ;` 的指令…… 不过好在这个编码也很简单，每个scoreboard对应一个bit，有就是1，没有就是0，合起来作为一个数就行。

线性关系不满足的情况

线性关系不满足的情况一般是汇编的文本上有一些隐含的规则，或是有一些特殊的处理。这类问题，我的一般处理方式是增加相应的自由度，让新的自由度承担这部分特殊的操作，然后让整个编码逻辑还能回到原始的框架内。

顺序相关的Modifier

当前我们假设所有的Modifier只与出现在opcode后或是第几个operand后有关，与它们在各自的operand中出现的具体顺序无关。大多数情况下这是没问题的。比如我们认为 `FADD.SAT.FTZ` 等价于 `FADD.FTZ.SAT` 。但是，对于有些指令，并不是这样。比如 `F2F.F64.F32` 表示F32转为F64， `F2F.F32.F64` 表示F64转成F32，两者显然是不一样的操作。实际上，Modifier出现在不同位置的指令（也包括出现两个相同Modifier的情况）还是有不少的，除了明显的 `I2I` 、 `F2F` 外，还有如：

```
IDP.4A.S8.S8 R9, R20.reuse, R25, R9 ;
HMMA.884.F32.F32.STEP0 R120, R196.ROW, R188.ROW, R120 ;
IMMA.8816.S8.S8 R36, R50.ROW, R74.COL, R36 ;
```

我这里处理的方法就是增加自由度把它区分开来。把每个modifier后相应的序号如 `0_F32@0` ， `0_S8@1` 等（前面的序号0表示出现在opcode里， `1_` 、 `2_` 等就表示出现在对应的operand里）。不过这里可能需要对这些指令进行相应的modifier筛选，把与位置有关的筛选出来。现在的观察来看，与位置相关的modifier基本都是data type（例如F32，I8，S32等），做一个查询表然后按出现顺序加后缀就可以了。

这里还有另一个有趣的问题，怎么知道一个opcode的modifier可能是顺序相关的呢？我这偷了个懒，做了点简化，按cuobjdump的逻辑，绝大部分modifier的顺序似乎都是固定的，只有顺序相关的才可能出现打乱的情况。这样我可以给每个opcode做一个有向图，opcode为根节点，把所有modifier也作为节点，然后把每个对应opcode的指令出现过的modifier序列按出现的顺序把它们用有向箭头连接起来。如果存在顺序相关的modifier，那这个有向图就会有环（这也可以检测同一个modifier出现两次的情况）。这样所有有环的opcode才需要加序号后缀（否则modifier分化太高，会大大增加复杂度和对输入的需求）。当然，我觉得肯定还有别的更好的方法，不过这个对我基本上够用了~

64bit格式转换指令

对于涉及64bit的格式转换指令（`I2F, I2I, F2I, F2F, FRND`，64bit类型包括F64、I64、S64等），即使汇编文本中的opcode是一样的，编码使用的opcode也会不一样。如：

```
I2F.F64.U32 R2, R4 ;          /* 0x0000000400027312 */
                              /* 0x001e220000201800 */
I2F.U32 R0, R4 ;              /* 0x0000000400007306 */
                              /* 0x000e620000201000 */
FRND.F64.CEIL R4, R4 ;        /* 0x0000000400047313 */
                              /* 0x002e620000309800 */
FRND.CEIL R7, R0 ;            /* 0x0000000000077307 */
                              /* 0x000ea20000209000 */
```

注意：Turing之后的格式转化指令中，类型为默认的F32和S32可能会省略，Maxwell/Pascal似乎是显示的。也就是说，这里`I2F.U32`隐含着`I2F.F32.U32`，`FRND.CEIL`隐含着`FRND.F32.CEIL`。可以看出`I2F.F64.U32`的opcode是0x312，而`I2F.U32`的opcode是0x306。单独用一个64bit的modifier其实不是什么问题，因为`F64`或`U64`的modifier完全有能力同时承担opcode和modifier本身值的改变（因为我们没有限定modifier的值必须是连续的bit）。但是如果同时出现两个64bit的modifier，那这个opcode的改变就会被加两次，那就会出问题。所以，对于所有涉及64bit的格式转换指令，我都显式的增加了一个opcode modifier（`0_CVT64`），这样它会承担opcode的改变，`F64`和`U64`这种modifier就只要承担自己那部分就行了。`FRND`因为只会出现一个64的modifier，加不加都无所谓。

这里其实看起来还有另一个问题，Turing的`I2F.U32`中其实隐含了一个`F32`的modifier，要加回去吗？其实不用，因为`I2F.F64.U32`中的`F64`会承担二者的差值。所以我不用显式的写`I2F.F32.U32`，实际`I2F`的值里就隐含了`I2F`+`F32`，而`F64`的值就相当于`F64`-`F32`，这样`I2F.F64`还是能与原始语义的值匹配上，线性关系不会被打破，所以这个modifier我不用自己去加也是可以的。这样，Maxwell和Turing我就不用做什么区别处理，这套逻辑都能用。当然，如果你想要解出每个modifier对应的bit值，那就补上即可。

PLOP3

`PLOP3`看起来像是Turing后添加的指令。`LOP3`（参考PTX文档：lop3）是3个32bit数的位运算，可以通过一个查找表（look-up table, LUT，PTX ISA里写作`immeLUT`）来指定3个输入之间任意的位运算组合。而`PLOP3`应该类似`LOP3`的逻辑，只是操作数变成了3个predicate。但是，如下指令：

```
PLOP3.LUT P2, PT, P3, P2, PT, 0x2a, 0x0 ;     /* 0x000000000000781c */
                                              /* 0x000fd80001f44572 */
```

其中的立即数查找表 `0x2a = 0b00101010` 在编码中不是连续的，而是形如 `0b 00101 xxxxx 010` 被其他的5个bit分成了两段，这是当前的逻辑无法正确处理的。所以这个立即数会被拆分成两部分的加权和。暂时我只看到这一个指令有出现这种立即数的bit不放在一块的情况。也不清楚是不是有什么内情？难道是从什么历史遗留指令改过来的？看起来也不像，存疑吧！

指令地址相关指令

我们前面讨论的这些指令都只与指令文本本身有关，与其出现的位置无关。但是，SASS中的很多跳转类指令，不仅和其汇编文本本身有关，还与该指令所在的地址有关。例如，我们可以对比一下这两条 `BRA` 指令：

```
/*0480*/              @!P0 BRA 0x500 ;              /* 0x0000007000008947 */
                                                   /* 0x000fea0003800000 */
...
/*04f0*/                   BRA 0x510 ;              /* 0x0000001000007947 */
                                                   /* 0x000fee0003800000 */
```

比如第一条指令文本中出现了 `0x500` ，表示跳转目标地址为 `0x500` 。但这是所谓的**短跳**指令，其立即数的编码方式都是目标地址相对于下一条指令的偏移值。因此，它编码所使用的值并不是 `0x500` ，而是 `0x500 - (0x0480 + 0x10)=0x70` （Turing每条指令长度为 `0x10` ）。同理，第二条 `BRA` 指令的立即数编码为 `0x510-(0x04f0+0x10)=0x10` 。但是，不是所有的跳转指令都用的短跳，**长跳**指令则可能会使用绝对地址。下面给出两个例子，大家可以仔细观察一下其中的差别：

```
/*0120*/          CALL.REL.NOINC R6 0x0 ;         /* 0xfffffed006007344 */
                                                   /* 0x013fea0003c3ffff */
...
/*0920*/          CALL.ABS.NOINC 0x0 ;            /* 0x0000000000007943 */
                                                   /* 0x001fea0003c00000 */
```

当前的线性逻辑不能有效的处理这些问题。但要加特殊处理其实也比较简单。如果是短跳（用相对地址），我就把后面的立即数改成对应的偏移值。如果是长跳（用绝对地址），那就不管。唯一有点烦的是需要判断某个指令是不是用了地址做操作数，还要确认用的是长跳还是短跳。好在这种指令也不多，问题也不大。

【注】：这里的短跳长跳我这可能用的不太严谨，大家理解就好。

这里附带提一句，这里出现的 `0x0` 其实不一定是跳转的目标地址。因为这是 `cuobjdump` 的输出，很多额外的信息没有显示出来。如果看 `nvdisasm` 的输出，则可以看到原始的语义是类似这种：

```
CALL.REL.NOINC R6 `(_Z4testPiS_S_) ;
...
CALL.ABS.NOINC `(vprintf) ;
```

其中 `_Z4testPiS_S_` 、 `vprintf` 是cubin中的**symbol**，其地址在编译期不确定，所以先填0（类似所谓的**fixup**），直接dump会出来0x0。这些指令在载入cubin时，这些symbol会得到实际的地址。驱动会按照**relocation** section（以 `.rel` 开头）的指示，根据相应symbol的实际地址修改掉原始指令的值。这样就可以跳到对应symbol的位置去。Cubin的Relocation是个大坑，将来讲ELF结构的时候再说，有很多问题我也没太搞明白…… 需要注意的是，无论是GPR的分配还是跳转的逻辑，它和直接inline的函数感觉也差别不大，只是代码没放在一起而已。不过，在有合适的ABI支持的情况下，也许可以实现多个kernel公用一部分代码？这就有点往CPU的函数调用逻辑靠拢的意思了。具体我暂时也没仔细研究~

隐藏指令

前面已经说了，我写汇编一般是基于CUDA C或PTX生成的cubin做修改。有些特殊情况下，NV的官方工具可能不会开放显示一些特定的指令汇编文本。当然这也许是bug，但不管是不是，总还是可能会有这个问题。这种情况下，我直接使用 `UNDEF 0x###` 这种方式来代替，其中 `0x###` 就是原始的指令编码。这有时候挺烦的，不过反正这部分我只管复现，如果不改它，问题也不大。不过将来如果需要做语义上的一些分析，比如自动生成control code，自动计算GPR数目等，这都是个负担。所以碰上估计还是只能去报bug，能走官方路子我总是欢迎的。

这种方式其实就类似LLVM的汇编器中直接用 `.byte` 在指令流里写入二进制值。有什么新的汇编器不支持的指令，也能这么hack，好处就是不用改汇编器。这当然是有隐患的，但是有时候作为一个makeshift，也算是一种解决方案。毕竟大厂修bug也是比较看缘份的……

小结

通过这些特殊处理（虽然有些处理很粗糙……），除了带QNAN的FSEL以及隐藏的汇编需要在反汇编时做改动，其他都可以直接从汇编文本做编码。对于Turing架构，我用官方库的文件做过测试，基本上所有的指令都可以复现。Maxwell和Pascal我还没仔细测，感觉上问题不太大。有些处理可能与Compute Capability有关，所以这部分需要与版本挂钩。但从总体流程来看，这套流程通用性还是比较强的，与版本关系不太大，人力因素介入也很少，要扩展到新的SASS指令集应该也问题不大。新的Ampere的指令我也跑过一些简单的测试，暂时还没看到什么问题，感觉上是能直接支持的（当然它和Turing差别也不太大）。

最后，前面浮点的坑没怎么太细讲，附一个浮点格式的文献，大家有兴趣可以自己研究：

[1] Goldberg, David. What every computer scientist should know about floating-point arithmetic[J]. Acm Computing Surveys, 1991, 23(1):5-48. 网上有重印的pdf版，强烈推荐对float格式和各种坑感兴趣的同学过一遍。