CUDA微架构与指令集（3）-SASS指令集分类

https://zhuanlan.zhihu.com/p/163865260

None Mon May, 24 03:20

今天聊一下CUDA SASS指令的基本分类。我手边只有一块笔记本的850M，是Maxwell的架构（与Pascal几乎一样）。之前的一段工作对Turing架构用的比较多，研究可能细一些，但现在手边没有Turing的卡可用了。所以这里主要会聊Turing的架构和指令集，顺带包括Maxwell和Pascal。但如果要做micro benchmarking，就暂时只能用Maxwell了。当然，虽然CUDA的SASS一直在演变，但从功能上也还是继承为主，所以这里谈到的多数内容，对其他架构也是适用的。这篇文章基本不直接涉及SASS的编码问题，主要讨论的还是SASS指令的功能分类，所以行文中多数时候用助记词（比如`FFMA`，`BRA`等）来代替编码。另外，同样的`NOP`指令，尽管它在不同的架构里编码不一样，在不引起混淆的情况下，我们还是把它当成同一个指令来讨论，毕竟功能上是一致的。

声明：从这期开始，就要涉及到很多硬件细节问题。坦率的说，很多东西都带有很强的“逆向”性质。不过这里的多数信息有的来自网络（也包括NV官方论坛），有的来自我自己的研究或是推测，无意去触及任何的商业秘密。我们还是以学术研究为主，最终目标还是更好的挖掘硬件潜能。所以如果有侵权或是涉及敏感信息，请联系我~ 不过我也相当于变相帮NV做推广了，我不收你广告费，你也别埋怨我说太多……

为了讨论方便，我随便在一个`cuobjdump`的`sm_50`(Maxwell架构)的sass中找了几行，作为例子：

```
                                        /* 0x001fc400fe4007fd */
/*0408*/        FFMA R2, R4, R2, R5 ;   /* 0x5980028000270402 */
/*0410*/   @!P0 FADD R2, RZ, -R2 ;      /* 0x5c5820000020ff02 */
/*0418*/        STG.E [R8], R2 ;        /* 0xeedc200000070802 */
```

真正的sass机器码写在每行最后，如`/* 0x001fc400fe4007fd */`，当前每行都是64bit。前面的`/*0408*/`是该行指令的地址（零点是当前kernel的起始位置，单位是Byte）。Maxwell架构是定长的64bit指令，所以每行对应一个指令。没有指令地址和文本的行，是编码control codes的行，具体将来再讲。

另外贴一个`sm_75` Turing架构的dump文件做对比例子：

```
/*0100*/        FSETP.NEU.AND P0, PT, |R10|, +INF , PT ;  /* 0x7f8000000a00780b */
                                                          /* 0x000fd80003f0d200 */
/*0110*/   @!P0 BRA 0x4c0 ;                                /* 0x000003a000008947 */
                                                          /* 0x000fea0003800000 */
/*0120*/        IMAD.SHL.U32 R4, R10, 0x100, RZ ;         /* 0x000001000a047824 */
                                                          /* 0x000fe200078e00ff */
```

形式和前面基本一致，但是由于Turing每个指令是128bit，所以每两行只有一行有指令文本。但是连续的两行都有参与到指令操作的编码中，第二行不都是control codes。

称呼问题及总体特征

聊指令集有一个很别扭的地方，就是有些东西没有官方或是比较众所周知的称谓，导致有些东西描述困难。所以为了行文方便，我这里首先统一一下我这里的称呼问题。这个东西其实挺主观的，有的是习惯性称呼，有的没见到别人专门讨论过，所以我就随便起了个名字……大家知道意思就好。如果有更好或是更广泛介绍的叫法，也请大家提出来~

首先，最简单的指令形式如`FFMA R2, R4, R2, R5 ;`。其中`FFMA`表示这个指令进行的是32bit float fused multiply-add 操作，也就是`R2=R4*R2+R5`。通常称`FFMA`为**操作码**（**Opcode**，当然它本质上还是指的`FFMA`对应的编码），后面的`R2, R4, R2, R5`表示参与操作的通用寄存器（General Purpose Register，简称GPR，有时也直接叫Register），称为**操作数**（**Operand**）。SASS中的习惯是第一个是目的操作数（`dst`），后面是源操作数（`src`，可以依次叫`src0`，`src1`，`src2`…或者`srcA`，`srcB`，`srcC`…都有见过）。

另一个指令`@!P0 FADD R2, RZ, -R2 ;`中出现了`@!P0`，`P0`是1bit的bool型的**谓词**（**Predicate**），`!`表示取反。指令前面加`@!P0`，表示谓词`P0`为否时才真正执行该操作，否则什么也不做。另外，这里其中一个操作数`-R2`出现了负号，有些地方称为**modifier**（不知道怎么翻译，修饰子？），相当于是修改了原操作数的含义。

比较复杂指令如`FSETP.NEU.AND P0, PT, |R10|, +INF, PT;`，`FSETP`是前面说的opcode，后面`P0`是目标操作数，`PT`是恒为`True`的Predicate，`|R10|`是`R10`加绝对值的modifier，`+INF`是一个float32的立即数，那`.NEU.AND`算是什么呢？如果把指令看成一个函数，opcode是函数名，operand是参数，那这些跟在opcode后的就是控制opcode一些具体操作细节的开关。具体来说，`FSETP`是一个通过比较两个float来设置predicate的指令，这里`NEU`指的是float比较中的Unordered Not-Equal（如果操作数有`NAN`则返回`True`，普通的ordered比较则只要有一个`NAN`就返回`False`，这是IEEE规范的做法）。`AND`表示比较完了再加个`AND`后得到最终结果。`.NEU.AND`这种在同一个opcode下再细化的，我也把它叫**modifier**，但这个我称为opcode modifier，而把修改operand的叫做operand modifier。后面我们会看到，其实两者的界限很模糊。反正我也不知道其他名字，所以先这么叫好了。一般opcode对应什么操作相对来说比较好猜，但每个modifier具体什么意义就比较困难，相当部分都很难搞清楚，这也是SASS学习中的主要困难之一。

这里再提一下研究SASS指令集的方法。一般说来，看到一个不明白的指令，总是会先搜一搜PTX有没有能直接映射到这个指令的伪指令，如果有，那就可以直接参考PTX的文档。如果没有或是形式上有些差别，那就需要写一些小测试程序，写一个已知的功能来触发这个指令，然

后通过理解整个汇编流程来倒推这个指令的含义。这个算是典型的逆向了。当然，有些功能也许不是在指令这一层实现的，有的是在driver中实现的，而driver这层一般对用户来说是透明的，那这个就很难研究了。

我大概总结SASS ISA有这么几个总体特征：

1. 指令长度定长。Volta以前几代都是64bit，而Volta、Turing、Ampere都是128bit。虽然有些架构的一些指令是没有实际操作的control codes，但每个有实际意义的指令都是按标准长度完全对齐的。而且多出来的control code也一定是一个指令的长度。
2. 基本算是load/store型的ISA，但是也有例外。除了constant memory外，其他memory只能在load/store型指令里做操作数，也就是必须要load到GPR里才能使用。constant memory是唯一可以作为非load/store指令操作数的memory类型。在Ampere之前，也没有直接的memory到memory的操作（必须memory到GPR，或是GPR到memory）。Ampere加了global直接到shared memory的操作，我还没仔细研究这是不是个例外。
3. 肯定不是CISC，但也不是典型的RISC。作为load/store型的ISA，那肯定离CISC是比较远的了。但它离那些简洁明快的RISC指令集似乎也比较远。很多SASS指令都支持非常复杂的、混合的操作，操作数多，操作逻辑也很复杂，功能上多样性很高。最多算是非典型的RISC。
4. GPU架构的控制类型的指令相对效率肯定是低一些。而且SASS指令集里只有很有限的控制指令（主要是分支和跳转），一些更复杂的辅助指令如debug和trap类指令只有在特定的debug程序段里才会出现，而且这类指令几乎是没有高性能模式的。

然后下面我们开始具体讨论每个指令部分。

再强调一遍：这里多数地方没有提到架构问题。因为虽然每代架构不太一样，但反汇编后的文本在语义上还是有很大的继承性，所以这里我主要谈反汇编后的指令含义，具体与架构的关系暂时放在一边。一般说来，如果某个架构的指令被废弃，那下一代一般也不会复用这个助记词，除非两者功能几乎一样。

Predicate

Predicate有时候代指指令前加的形如 `@P0, @!P3` 等的predication，PTX文档里也把这个叫guard predicate。有时候也会特指那个predicate register，如 `P0, P6, PT` ，反正都差不多这个意思，不混淆就行。在之前的文章也提到过，在现有的所有架构中，每个指令都有4bit的编码来指定每个predicate，3bit用来指定索引（所以每线程有2^3=8个predicate register `P0~P7` ，其中 `P7=PT` 为恒 `True` ），1bit表示是否取反。如果是 `@PT` ，那就会在反汇编中省去不显示。那如果是 `@!PT` 呢？嗯，大家自己想……

Predicate是控制某个线程是否执行某个指令的两种方式之一，另一种就是conditional branch。两者的区别在于用predicate时，可以让warp内的所有线程名义上走同一路径而省去跳转的开销，从而也避免了可能的divergence。因为branch的latency比较长，还涉及到instruction cache的问题，一般很短的分支是不太愿意跳转的。当然branch在出现divergence的时候，内部也有一个mask，表明当前这个thread是否active，但是用户不能直接修改这个mask。PTX中可以通过warp vote或是load特殊寄存器 `%lanemask_*` 之类的方法获得当前warp内的mask情况。

Opcode 和 Opcode Modifier

opcode是指令的大类，表明该操作的主要工作形式和操作对象。具体支持的opcode可以参考 Instruction Set Reference。当然光看这个表几乎等于没看，只有最简单的功能介绍，连编码和操作数介绍都没有，我甚至怀疑有些地方还有问题。但这也是唯一的官方介绍了，凑活看吧！我对指令集的研究也很不全面，下面简单列一下我的归类。

Float指令

基本分为4大类：

- float64，以D开头，如加法 `DADD` ，乘法 `DMUL` ，乘加 `DFMA` 等。
- float32，以F开头，如 `FADD` ， `FFMA` ，最大最小值 `FMNMX` 等（英文其实挺别扭的，两个比也有“最”）。
- float16，以H开头（Half），如 `HADD2` ， `HFMA2` ，比较 `HSET2` 等。
- `MUFU` 指令，包括所有的特殊函数指令（SFU中执行的指令）。比如倒数 `rcp` ，倒数平方根 `rsq` ，对数 `lg2` ，指数 `ex2` ，正弦 `sin` ，余弦 `cos` 等等。具体支持的函数列表可以参考CUDA Math API中instrinsic functions，当前其中很多并不直接对应到一条指令，而且有些是普通float指令，并不是SFU执行。注意 `MUFU` 指令都是近似算法，精度可以参考 intrinsic functions。我印象中 `MUFU` 应该是有F32和F64的指令，其中F64应该只有 `MUFU.RCP64H` 和 `MUFU.RSQ64H` 这种指令，且精度与F32相当。F16暂时还没见到过，这个有待仔细研究。

float指令的几个比较值得注意的点：

- `FMA（d=a*b+c）` 可以在一条指令里一次性计算乘和加，与用两个单独的乘法和加法相比，不仅速度快（一般说来 `FMA` 和 `MUL/ADD` 的throughput一样），而且精度还更高（两次round变成一次）
- 没有直接的除法指令。浮点除法开销很大， `x/y` 的近似算法是用 `x * rcp(y)` 来算的。精确算法一般是需要 `rcp(y)` 得到初值后，进行多步迭代。所以浮点数除法是比较慢的操作。
- 一个GPR是32bit，可以放两个F16，所以H开头的指令一般都是 `H*2` 的形式，可以同时算两路（类似SIMD）。

- 多数float指令都支持一些opcode modifier，比如四种round的模式（最近偶数 `RN`，趋0 `RZ`，趋负无穷 `RM`，趋正无穷 `RP`），是否flush subnormal（`FTZ`，flush to zero，把指数特别小的subnormal数变为0），是否饱和（`SAT`，指saturate，将结果clamp到[0,1]之间），等等。operand modifier主要就是取相反数（`-R0`）或取绝对值（`|R0|`），也可能一起用（`-|R0|`）。

Integer指令

这是最常用也是最主要的指令分类。大致可以细分为以下几类：

- 算术指令：如加法 `IADD`，`IADD3`，乘法 `IMUL`，32bit乘加 `IMAD` 或是16bit的乘加 `XMAD`（Maxwell或Pascal）。还用一些特殊算术指令，如 `ISCADD` 和 `LEA`，两者与 `IMAD` 很相似，但语义不同。还有如dot-product的 `IDP/IDP4A` 等等。
- 移位指令：如左移 `SHL`，右移 `SHR`，漏斗移位（Funnel Shift）`SHF` 等等。
- 逻辑操作指令：现在多数逻辑操作都用3输入逻辑指令 `LOP3` 来实现，它支持三输入的任意按位逻辑操作。这个指令很灵活，也有很多妙用，将来会再细讲。有兴趣的读者可以先看看PTX的介绍:lop3。
- 其他位操作指令：如计算1的个数 `POPC`，找第一个1的位置 `FLO`，按位逆序 `BREV` 等等。这些指令在一些特殊的场合下会收到奇效，特别是在一些warp内的相互关联操作，能形成很精妙的配合。
- 其他：比如 `IMMA`，`BMMA` 这种Tensor指令。

整数指令里面几个值得关注的地方：

- 整数乘法的实现。通用的32bit乘法或乘加，除了Maxwell和Pascal中用的是 `XMAD`，Kepler和Volta、Turing、Ampere都是用 `IMAD`。但是很多地址计算中有这种模式：`d=a*Stride+c`，在 `Stride` 是2的幂次时，可以用移位和加法来实现。这正是 `LEA` 指令的工作模式。Turing的 `IMAD` 和 `LEA` 分属不同的dispatch port，两者可以独立发射。因此这是一个可能增加ILP的小优化。
- 整数的除法和取余并没有独立的指令。当除数在编译期未知时，需要几十条指令来做整数除法和取余计算。但是当除数已知时，可以通过乘以一个magic number后再移位来快速计算，这样只需要很少几条指令。这个一般编译器可以识别到，会自动优化。
- 整数的移位有三种操作，左移、右移和funnel移位。但是并没有区分有符号数的逻辑移位和算术移位，这个在opcode modifier中可以指定。注：理论上只有右移才区分，逻辑右移补0，算术右移补符号位。左移反正都是补0，负数最高位是0的话反正也要溢出，就无所谓了。另外，如果移位的量是编译期常数，其他一些指令也可以用来组合成移位的效果，比如 `IMAD` 和 `LEA` 等等。
- Turing的 `IMAD` 是个挺神奇的指令。大量的情况下会用来做 `MOV` 操作，比如 `IMAD.MOV.U32 R1, RZ, RZ, R0;` 的作用就相当于 `MOV R1, R0;`。那好处在哪呢？这个应该与Turing把

Float32与普通ALU的dispatch port分开有关，`IMAD`用的也是float32的pipe，所以可以与`MOV`的发射错开，这个到聊指令发射逻辑的时候再细讲。`IMAD`还有带shift的模式，如`IMAD.SHL.U32 R0, R0, 0x10, RZ ;`，还有`IMAD.WIDE`可以用64bit数做第三操作数，等等。

格式转换指令

这主要是数值格式的转换。因为cuda里带格式的GPR其实就两大类，整形(I)和浮点型(F)，所以格式转换主要就四个指令：`I2F, I2I, F2F, F2I`。然后opcode modifier会具体指定整形和浮点型的位数（主要是16，32，64），整形还分有符号无符号等。注意`I2F`和`F2I`中opcode modifer中I类型如果不写就默认是`S32`，F类型不写就默认是`F32`。比如`I2F.F64 R12, R13 ;`就是把S32的`R13`转为F64的`R12`，`I2F.U32 R15, R12 ;`就是U32转F32，都不写类型的如`I2F R19, R14 ;`，就是S32转F32。如果src是float型，一般还有modifier来指定取整方式，如`TRUNC`。如果是`F2F`和`I2I`，则两者类型在opcode modifier中都要写，因为和顺序有关。

格式转换里还有一个`FRND`指令，是浮点到浮点的取整转换。【注】：我粗粗研究了下，volta之前的架构好像用的F2F.F32.F32.FLOOR/TRUNC这种格式，Volta后才加的FRND。这个感觉在自己做线性插值时很方便，直接用f-FRND(f)就可以得到f的小数部分，不用转int。

数据移动指令

这一类就是各种数据搬运了。最典型的就是`MOV`了。不过需要强调的是，GPU的GPR并没有类似x86的`eax`、`ebx`之类那么多的隐式含义，所以GPR到GPR的`mov`多数情况下是不需要的（当然也有一些例外）。`MOV`指令多数时候的用法是把一些立即数或是constant memory甚至是0，移动到GPR里。但是后面也会讲到，首先这些大多数可以直接做operand，二来往往有一些特殊的指令可以帮忙，比如设置0可以用`CS2R R0, SRZ;`。搬运constant memory的方法就更多了。而且前面也讲了，`IMAD`也可以做`MOV`，还可以不占ALU的dispatch端口。所以一般优化后的程序中`MOV`并不常见，如果很多，说明优化上还是有一些问题的。当然，debug版本往往满屏的`MOV`，另当别论。

移动指令还有byte的permute指令`PRMT`，符号扩展指令`SGXT`等。这一类中有一个特别重要的指令是warp shuffle指令`SHFL`。warp内如果需要进行数据交换，第一要想到的就是这个指令。它支持多种交换模式，对其他warp没有依赖，因而在一些场景下有很大的用处。其中一个典型应用是做warp内的reduction，比如scan（或者叫prefix sum）之类。有兴趣的读者可以看看cuda sample里`shfl_scan`这个例子。`SHFL`和warp投票指令`VOTE`，还有前面说到的`POPC`和`FLO`，可以做一些奇妙的组合，这个将来有机会再展开讲。不过Ampere里加了一个`REDUX`指令，可以直接做warp的reduction，不知性能如何，我还没怎么研究。

Predicate操作指令

Predicate既然是1bit的bool型，那逻辑操作`PLOP3`就当然是有的。但是它的操作数特别多，形式如：`PLOP3.LUT P1, PT, P1, P2, PT, 0xa8, 0x0 ;`或者`PLOP3.LUT P0, PT, PT, PT, PT, 0x8, 0x0 ;`这种。这个东西我也没研究清楚，总之比较复杂，有些操作数就没看到不是`PT`的情况，所以也不知道具体是什么意思。

前面说过，Predicate一个线程可编程的只有7个，如果不够用，可以把Predicate转存到GPR里，这里就需要`P2R`和`R2P`指令。

内存操作指令

内存操作指令比较复杂，也是重点需要关注的指令类型。除了通常意义上的各个内存的load、store指令外，还有texture和surface的操作指令。这里按操作类型简单分个类：

- memory的load操作：根据memory所属域的不同，Generic就用`LD`，global用`LDG`，local用`LDL`，shared用`LDS`，constant用`LDC`。如果有tensor load的功能，还有`LDSM`可以直接load matrix。
- memory的store操作：与load对应，Generic用`ST`，global用`STG`，local用`STL`，shared用`STS`。由于constant memory是只读，就没有store了。
- memory的atomic操作：所谓的atomic操作一般都遵循read-modify-write的流程，常见操作有Compare-And-Swap（CAS），Exchange，Add/Sub（或者加减一Inc/Dec?），Min/Max，And/Or/Xor等等。根据对象的不同，Generic用`ATOM`，global用`ATOMG`，shared用`ATOMS`。constant只读，所以没有atomic操作。local memory是私有的，没有线程竞争，所以也没有atomic操作。注意不是所有类型的所有操作都有相应的指令。因为这些计算单元多数无法复用现有单元，所以开销很大。因此很多操作是在软件层用CAS循环的方式实现的，并没有相应的指令。atomic有需要返回值和不需要返回值的形式，如果不需要返回值，那就称为reduction，用`RED`指令。当然，你把返回值写到`RZ`可能是一个效果，存疑。
- Cache control指令：这个主要意义是我知道我将要访问的某个元素位于某个cache line，但是又不确定具体要哪一个值，所以没法先load。所以可以先把整个cacheline放到cache里，等到要用的时候从cache里取。这样load的latency可以更好的被隐藏。PTX有相关的控制指令，但是我在Turing上测试过几次，只测到过TLB的latency被消除，实际访问latency好像没变，存疑。
- Texture操作指令：Texture因为在kernel内是只读的，所以只有load。`TEX`应该是只fetch不插值（只有1D有fetch），相当于一个load功能。`TLD`和`TLD4`应该是可以做插值。Texture在HPC应用也还是有一些，但用到的功能并不像图形应用那么多而复杂。所以这个我也不太熟悉。

- Surface操作指令：Surface相当于可读可写的Texture，但是没插值之类的功能。两者还可以bind到同一个 `cudaArray` 上。surface的load相当于texture的fetch，用整数输入，不插值。不同的是texture的fetch应该是只针对1D，而surface可以有2D，3D。也就是说，如果是1D的array可以用texture的fetch，2D、3D就只能用surface的load了。具体指令有surface的load(`SULD`)，store(`SUST`)，atomic(`SUATOM`)和reduction(`SURED`)。

memory由于功能上的多样性很高，所以几乎每个指令都有很多opcode modifier。常见的比如load可以是32bit，64bit，128bit。global的load还可以指定是不是过L1 cache（ `CONSTANT` ）。其中很多还可以指定memory consistency对应的model和scope。这都是非常复杂的问题。不过好在大部分内容PTX里都有提及，文档也比较详细，有兴趣的读者可以自行研究。

memory类的指令是性能优化的重点中的重点。绝大部分未经优化的程序都会是memory bound，多数实际应用优化完了还是memory bound。所以，通过合理的选择相应指令，搭配合适的内存排布，从而更好的隐藏内存访问的latency，或者是减少相应访问的开销，是性能优化中的主要课题之一。

跳转和分支指令

跳转和分支指令是SASS指令集中随架构变化最频繁的指令。虽然不清楚硬件具体做了哪些改动，但在指令的选择上就可以发现一些变化。不同的架构里经常看到一些形式上特别接近但又不完全一样的助记词。比如Maxwell里有 `BRK, SSY, SYNC, CAL` ，而Turing里有 `BREAK, BSSY, BSYNC, CALL` 。前面我们说过，如果功能一样，助记词应该会复用的。但名字不一样，说明功能上还是有明显变化的，所以才要换个名字以示区分。我可以简单的列一下两种架构的一些典型跳转和分支指令的形式：

Maxwell:

```
BRA 0xe80 ;
BRX R4 -0x1a80 ;
PBK 0x2598 ;
BRK ;
SSY 0x2ec0 ;
SYNC ;
CAL 0x48f0 ;
PRET 0x760;
RET ;
```

Turing:

```
BRA 0x210 ;
BRX R10 -0xe50 ;
BMOV.32.CLEAR RZ, B1 ;
BREAK B3 ;
BSSY B0, 0x3c00 ;
BSYNC B1 ;
CALL.REL.NOINC 0xbe10 ;
RET.REL.NODEC R6 0x0 ;
```

由于PTX往往隐藏了绝大多数的跳转实现细节，所以我们很难得到跳转指令的一些具体信息。我这里只谈一下简单概念，以后再仔细研究讨论。我把跳转和分支指令分为这几类：

- **定向跳转或条件定向跳转**：最典型的就是`BRA`，如`BRA 0xe80 ;`。这类跳转的目的地是确定的（目标操作数是立即数），但是可以跳或不跳。SASS里并没有直接的条件跳转指令，所有的条件跳转是用predicate实现的。
- **不定跳转**（indirect branch）：最典型的是`BRX`，如`BRX R10 -0xe50 ;`。这里跳转的目的地是不确定的，可以由GPR来指定跳转位置。当然，这个也可以加条件。不定跳转的好处是可以支持类似函数指针等列表形式的目标，编程上具有更好的灵活性。如果是`BRA`是`if-else`，那`BRX`就相当于`switch-case`。
- **分支管理操作**：前面也说了，条件跳转是用predicate实现的。那什么时候会出现warp divergence其实是不确定的。而且是否出现warp divergence当前线程不能直接知道。所以SASS里提供了一些分支同步的点，显式的做branch的synchronize和converge。典型的就是Turing的`BSSY`和`BSYNC`（Maxwell用`SSY`和`SYNC`）。Turing里还有`BMOV`之类，感觉是用某个Barrier做分支同步，但也不知道具体是怎么做的，待研究。
- **跳转目标管理**：`BRA`指令是单纯的跳走，相当于没打算跳回来，或者是有确定的目标可供下一次跳转。而有些跳转前要做准备工作，将来可能会跳回来继续执行下面的指令，这是函数调用最常用的形式，所以需要保存当前的位置（CPU里通常要把context压栈，但GPU的context一般不动，只保存PC）。Maxwell里跳转前都需要用`PBK`或是`PRET`之类(就是pre-Break, pre-Return)保存回跳位置（也许是存在某个隐式跳转栈里面）。而到了Turing里面，这个隐式的栈被去掉了，直接显式的用GPR保存回跳位置，然后`RET`或是`BREAK`的时候把这个GPR作为操作数就行了。这也体现了当前SASS设计演变的一个趋势，或者说是其设计理念：显式优于隐式。
- **特殊跳转指令**：比如`EXIT`。还有一些break point或是trap handler之类。这些一般都是在特定功能下才启动，比如debug之类。

跳转是个很复杂的问题。GPU中对跳转的处理也有很多值得研究的地方。不过具体指令上，有时候也没那么重要。更多的时候我们只是关心什么时候会有divergence导致性能损失而已。所以这些东西知道就好，一般也不用过于深究。

其他辅助指令

主要是一些常用（或是不常用……）的辅助指令，比如：

- `BAR`，barrier同步指令。一般是block内的某warp的内存读写需要被其他warp看见时用。
- `S2R` 和 `CS2R`，把特殊register的值载入到GPR。常用的特殊寄存器有：`SR_TID.X, SR_TID.Z, SR_TID.Z, SR_CTAID.X, SR_CTAID.Y, SR_CTAID.Z, SR_LANEID`，`SR_*MASK`（`LT`，`LE`，`GT`，`GE` 等），`SR_CLOCKHI/SR_CLOCKLO` 等等。具体可以参考PTX关于特殊寄存器的文档。我记得 `CS2R` 与 `S2R` 的区别是 `CS2R` 的latency是固定的，`S2R` 则不固定。不过 `CS2R` 好像就只看到支持 `SRZ`（就是 `RZ`？）和 `SR_CLOCKHI/SR_CLOCKLO`。这里附带提一句，`blockIdx` 对应 `SR_CTAID`，`threadIdx` 对应 `SR_TID`，两者的xyz都有special register，那 `blockDim` 和 `gridDim` 呢？其实很简单，只有变动或是不确定的数才有放到special register内的价值，`blockDim` 和 `gridDim` 在kernel运行时都是全局定值，现在都会被放在constant memory里。编译器会自动把相应访问转为对constant memory相应地址的访问。
- `DEPBAR`，这是线程内依赖检查的指令。这里涉及到control codes和依赖计数器的问题，有机会再讲。
- `NOP`，啥也不做，就占个位置，CPU里常用类似 `move eax, eax` 之类的方式。但GPU就显式的用 `NOP`。每个kernel最后一般也会用这个指令做padding，从而让指令section满足对齐要求。
- `VOTE`，这是warp的投票指令。一个warp有32个线程，每个都 `True` 或 `False`，正好可以组成一个32bit的数。
- 还有其他一些不怎么常用，代码里也不怎么见得到。碰到了再说吧！

Uniform DataPath指令

从Turing开始，SM里加入了一类新的ALU功能单元，用来进行warp内的一些公共计算。与此相配套的，自然就有uniform datapath的指令，还有Uniform的register和preidcate。如果说其他指令是32 lane的vector指令，这个就是1 lane的scalar指令。Uniform datapath的指令是针对warp而言，相当于每个warp只需要单个执行即可。比如之前算b=a+1，那warp内每个线程都要有GPR来保存a和b的值，相当于要32份GPR，做32个操作（虽然是同步的）。但如果warp内所有的a都是一个数，那算出来的b肯定也是同样的，我们没有必要做32个同样的操作，直接用一个公用的功能单元把这个算出来，然后让大家都可见就行了。AMD的架构很早就有scalar ALU和相应的scalar GPR，不知道NV是不是受了它们的启发……不过AMD还有scalar memory指令，还有对应的scalar cache，用起来还是要更灵活一些。NV暂时还没有看到。

Uniform datapath指令只支持int的ALU指令，并没有float，也没有跳转，也没有memory访问（constant memory除外）。不过当前Uniform datapath指令由于没法从普通memory里读数据，也没有从warp里某个lane读数据的指令【勘误：R2UR好像可以】，所以它的数据来源非常有

限，灵活度很低。从当前来看它主要的用途，是计算warp的公共地址或类似的公共参数（比如基于blockIdx和constant memory的一些运算），这也是比较容易出现计算冗余的地方。至于具体的指令功能，与普通的integer指令几乎是一样的，区别主要就是操作对象。所以这里也不再赘述。

Uniform datapath也支持guard predicate，但显然没法依赖warp内某个线程的predicate register，所以也有uniform predicate register。 前面一般写 `@UP0` 这种，但是编码似乎用的是与普通predicate一样的形式。

当前PTX和CUDA C都没有为uniform datapath指令专门设计一个相应的编程模型，这些指令的选用是ptxas根据推导做出的选择。但将来，等设计成熟功能完善后，如果有一些相应的编程模型直接映射到相应指令，在一些问题的优化上还是会很有帮助的。

Operand 和 Operand Modifier

SASS的Operand主要分为这么几类：

GPR

GPR是thread内最常用也是量最大的资源。因此绝大多数的指令的操作对象都是GPR。SASS里通常都用 `R` 加一个十进制数来表示（前面不补0），例如 `R0`，`R17`，`R250` 等。现在稍微新一点的架构的GPR编码都是8bit，意味着最多可以编码到(`2^8-1=255`)。但是 `R255` 被指定为一个常0的GPR，称为 `RZ` 。所以每个线程最多可以用255个GPR。注意这是编码上的限制，实际用量是用户或编译器控制的。一个GPR是32bit，但有些指令需要更多的位数，这就需要占用连续多个GPR。SASS里不管是用的单个的32bit，还是64bit或128bit，都只会在反汇编文本里写第一个GPR。比如 `LDG.E.128.SYS R4, [R26] ;` ，其中 `R26` 是64bit的地址，所以实际是 `R[26:27]` ，而 `R4` 是128bit load的目标操作数，所以实际上用的是 `R[4:7]` 。这里还有一个隐式的对齐要求，连用两个GPR的话就需要第一个是偶数（GPR编号从0开始），连用四个的话第一个就一定是4的倍数。

一个指令最多可以支持4个GPR操作数（64bit或128bit也算一个操作数），三个src，一个dst，比如 `FFMA/DFMA/IMAD/LOP3` 等。用更多GPR操作数的我还没见过。3个GPR的src中有一个GPR可以被替换成后面的Constant memory、Immediate或是Uniform register，但只能替换一个。这里搬运一个之前用过的例子，对于 `FFMA` （ `d=a*b+c` ），有这些操作数的形式：

```
FFMA R0, R1, R2, R3 ;                          // R0 = R1*R2 + R3，全是普通GPR输入
FFMA R5, R10, 1.8446744073095516160e+19, RZ ; // a或b是立即数，a、b等价，可互换，所以一个支持就够了
FFMA R2, R10, R3, -1 ;                         // c是立即数
FFMA R5, R5, c[0x0][0x160], R6 ;               // a或b来自constant memory
FFMA R0, R5, R6 , c[0x0][0x168];               // c来自constant memory
FFMA R14, R19, UR6, R0 ;                       // a或b来自uniform register
FFMA R18, R16, R13, UR6 ;                      // c来自uniform register
```

注意 `RZ` 和普通GPR地位是一样的，没有个数限制，它甚至可以做dst，相当于抛弃这个输出。

GPR支持很多种形式的operand modifier，比如前面提到的float的取相反数（`-R0`）和绝对值（`|R0|`）。印象中整形也可以取相反数，绝对值相对少见，但是有按位取反（如`~R0`）。这些operand modifier都可以用到constant memory和Uniform register上。immediate用应该就意义不大，不过编码那些位在不在用没仔细研究。

Opcode有一些modifier其实是dst的operand modifier，比如是否饱和（`SAT`）；有些是src和dst通用的modifier，比如`FTZ`。所以前面也说过，opcode的modifier和operand的modifier其实界限比较模糊。

Predicate Register

控制指令运行的Predicate已经讲过很多次了，而Predicate Register做操作数跟做指令的predicate编码是相似的。总共4bit编码，其中3bit编号，对应`P0~P6`再加恒`True`的`PT`，还有1bit表示是否取反。predicate做操作数的场景还是比较多的，比如用做carry，用做比较操作的结果，用作`SEL、FSEL、IMNMX、FMNMX`等指令的条件判断输入，还有一些专门操作predicate的指令如`PLOP3`等。很多SASS指令都会加上predicate的操作数，但确实有很多并不太清楚具体含义和用法。

Predicate操作数在指令中的用量似乎很不受限制，比如`PLOP3.LUT P0, PT, PT, PT, PT, 0x8, 0x0 ;`其中出现了5个。它可以和大量GPR一起出现，比如`IADD3.X R27, RZ, R15, RZ, P0, !PT ;`其中用到了6个操作数，4个GPR，2个Predicate。Predicate还可以做dst操作数，比如前面的`PLOP3`，还有各种比较操作（`*SETP`）等等。应该说是用量很大…… 现在Turing的编码多达128bit，所以我感觉有些地方Predicate的使用似乎是有点过滥了，就是不知道硬件开销具体怎么样。

Predicate除了前面取否外，一般没有什么别的operand modifier。

Constant memory

总体来讲，SASS基本上还是load-store型的指令集。也就是说，memory操作数只会出现在内存访问指令中。但constant memory是个例外。多数ALU指令都支持把其中的一个本来是GPR的操作数换成Constant memory。不过好像有个限制，因为constant memory虽然是个常量，但在`LDC`指令中也可以用GPR索引的方式来访问，不同的线程如果传入了不同的索引地址，就可以取到不同的值。warp内请求的是同一个地址时效率最高，多个请求地址也支持，只是会被serialized，性能会下降。所以constant memory做ALU的operand时，似乎是不支持GPR索引的，这个我还没有仔细确认。

constant memory也支持32bit和64bit的模式，128bit好像比较少见，不太清楚。一般说来，取代原GPR操作数的constant memory可以支持相同的operand modifier。

一般constant memory的写法类似`c[0x0][0x10]`，前面中括号表示bank，后面表示地址。现在constant memory主要有两个来源，一是用户显式的用`__constant__`声明的，二是driver或编译器自动生成的，包括kernel的参数，blockDim，gridDim等等。这两个一般会放在不同的bank里。还可能有一类是编译器自动把一些编译期常数放到constant memory里，但这个与编译期选择有关，我不太确定。

Immediate

SASS主要有两种立即数，整形和浮点型。整形的位数和是否有符号是由指令决定的。浮点型比较有意思，当前支持三种浮点型F16，F32，F64。但是立即数最多支持32bit编码，所以F64的浮点数是放不下的。一般F64的立即数只保留了前32bit（注意还是用F64的格式，不是F32格式）。这样有些浮点值就不能精确表示了。很多F64的数学函数最后都需要一长串的DFMA来做多项式求值，很多常系数没法放到FFMA的立即数里。所以要么前面MOV到GPR，要么就直接放到constant memory里。

立即数多数没有必要支持operand modifier，因为多数情况下立即数本身就有直接编码这些数的能力。

Uniform Register和Uniform Predicate

UR和UP是Turing架构开始才有的操作数。当前UR编码是到64个，`UR63=URZ`也是恒0，UP和普通predicate的编码是一样的。UR因为资源开销并不大，感觉并没有按总量分，应该是见者有份，每个warp都可以用满63个。 UR和UP可以作为Uniform datapath指令的操作数，也可以做普通指令的操作数。但普通的GPR和predicate不能做Uniform datapath的操作数，因为用了可能就不Uniform了。

地址操作数

很多ISA并不会把所谓的地址作为一种操作数，因为反正每个域都是独立编码的。但是由于SASS里取址模式的多样性。这里也单独拿出来说一说。先列一波以壮行色：

```
LDG.E.128.CONSTANT.SYS R4, [R6] ;
LDG.E.64.SYS R32, [R10+-0x80] ;
LDG.E.SYS R11, [UR8] ;
LDG.E.SYS R4, [UR4+0x4] ;
LDG.E.64.SYS R12, [UR4+-0x8] ;
LDG.E.U16.CONSTANT.SYS R6, [R6.U32+UR4+0x2200] ;

LDS.U R3, [0x4] ;
LDS.U.64 R10, [R7] ;
LDS.U.64 R10, [R7+0x8] ;
LDS.U.U8 R12, [R8+UR4] ;
LDS.U.U8 R11, [R8+UR4+0x3] ;
LDS.U.U8 R15, [R8+UR4+-0x1] ;
LDS.U R5, [R3.X4] ;
LDS.U.64 R2, [R2.X8+UR4] ;
LDS.U.64 R6, [R7.X8+UR4+0x10] ;
LDS.U.128 R4, [R4.X16] ;

LDL R11, [R0+-0x8] ;
LDL R4, [R10+UR4+0x10] ;
```

首先可以看到地址里可以有GPR，有UR，有立即数（有正有负），三者基本上算是可以自由组合。global和shared还有有一些细微的差别，global的地址其实是虚拟地址，base一般是一个不确定的值。但LDS一般用的是实地址，base从0开始（相当于当前block分配到的第一个byte）。所以LDS支持 `R0.X4` 、 `R0.X8` 、 `R0.X16` 这种模式，相当于 `R0` 可以不用乘element的byte数了，能省下一些指令。另外，我还没有仔细研究过其中哪些是32bit，哪些是64bit。一般global的最终地址肯定要是64bit， `LDG.E.U16.CONSTANT.SYS R6, [R6.U32+UR4+0x2200] ;` 里应该UR4是64bit，R6是32bit，那是不是可以反过来呢？没仔细研究过。这个和一些地址计算还是有很大关系，因为64bit的整数运算开销还是比32bit要大不少。shared memory因为空间小，用32bit就足够了，所以肯定都是32bit的运算。local memory的用例比较少，我研究得不多，也不太确定。

其他操作数

还有一些不能归类到上述操作数类型的操作数。比如前面讲过的特殊寄存器 `SRZ, SR_TID.X, SR_CLOCKHI` 等等，还有一些内置的barrier等等，总之就是内部或外部一些可供读取的状态值。这里简单列几个：

```
CS2R R4, SRZ ;
S2R R20, SR_CTAID.X ;
S2R R5, SR_LANEID ;
BMOV.32 B6, R2 ;
DEPBAR.LE SB0, 0x0 ;
DEPBAR.LE SB0, 0x0, {2,1} ;

R2P PR, R98.B1, 0x18 ;
P2R.B1 R98, PR, R98, 0x18 ;
```

还有的可能都不算是操作数。比如：Texture访问指令 `TLD.SCR.LZ.NODEP RZ, R11, R4, 0x0, 0x5f, 1D, 0x1 ;` ，其中的 `1D` 算操作数吗？好像很勉强，也许作为opcode modifier更合适吧！不过这也不是用户定义的，他怎么写我们就怎么用。这些名称问题不用太纠结~

Control Code

从Kepler开始，NV把一些线程控制的逻辑编码到了指令中，每7条指令有一条control code指令。到Maxwell的时候，control codes就相对比较完善了，每3条有一条。Turing没有改变control codes的内容，但是由于指令长度变长了，所以可以每条指令自己编码。

Control code主要有reuse、read barrier、write barrier、wait barrier、yield hint、stall count等几个域。reuse是唯一能在反汇编文本里看到的。它可以有限的解决一些GPR的bank conflict问题，也许同时还能减少GPR的读写，节省一点功耗。关于barrier的几个主要控制thread内的依赖性问题，这会影响指令发射的仲裁过程。yield和stall count主要影响的是warp调度的逻辑。这些牵涉到非常多微架构的细节实现问题，下次专门讲指令发射和warp调度逻辑的时候再展开讲吧~

这次林林总总讲了这么多，很多东西我也不是很确定，抛砖引玉吧~