**1 算法分析**

**1.1 主算法流程**

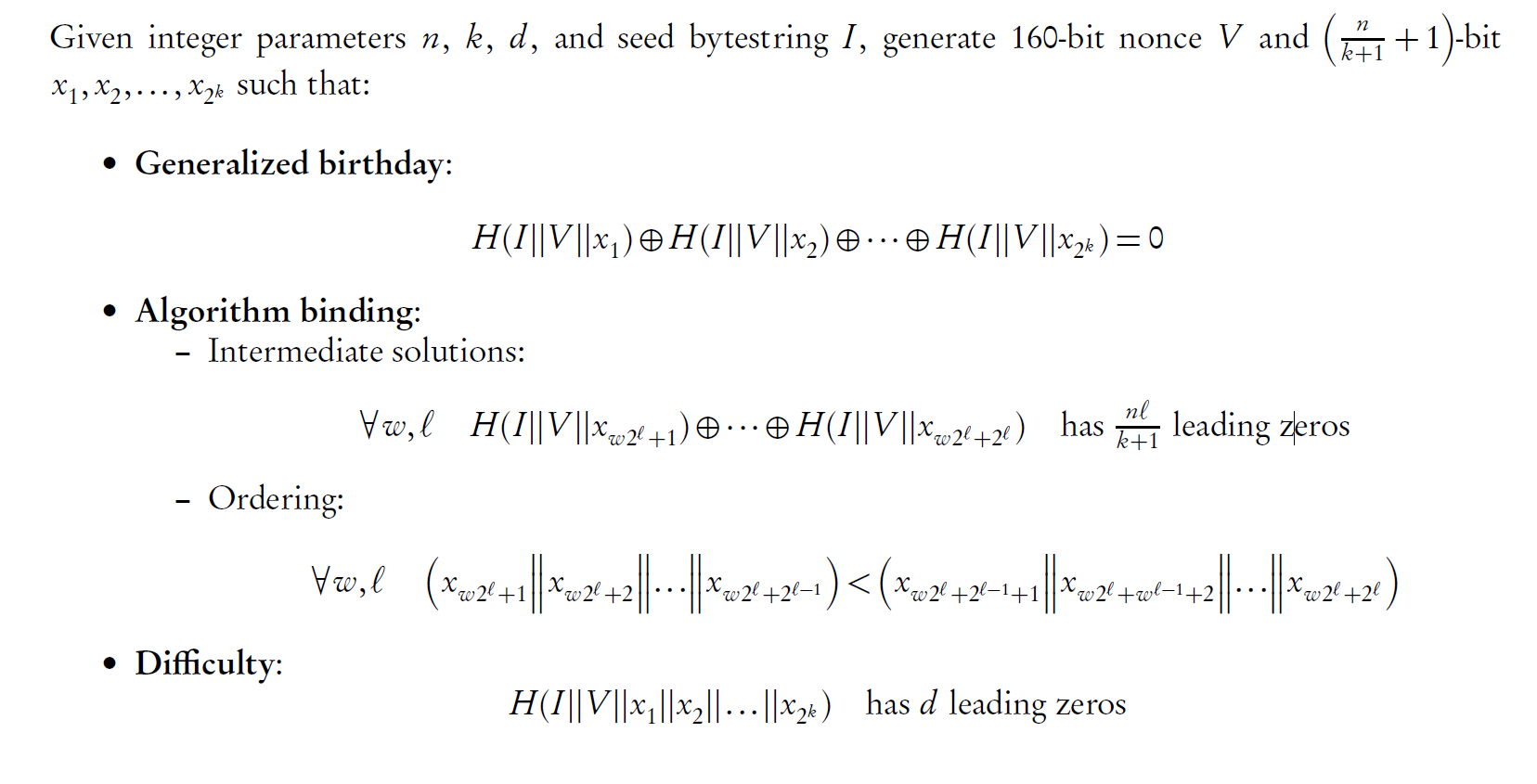
Equihash处理主要分为3分部分包括：

1）数据集产生，利用BLAKE2b算法

2）在数据集中寻找碰撞数据

3）验证碰撞数据结果并获得结果值

其中比较复杂的是在数据集中寻找碰撞数据，该问题属于广义生日问题（GBP），并且加上一些搜索限制条件，整个问题描述为



算法按照步骤执行，根据程序理解，主要步骤包括

1. 产生数据集
2. 顺次执行n/(k+1)bit（20bit）的碰撞（从第二次开始，第一次值记录下来最后作），对找到的碰撞结果记录碰撞索引号及相对碰撞步进，并计算保存下一次需用的xor值
3. 重步骤2）复并完成8次计算
4. 第9次计算对碰撞对象反向推演出序号，并计算第一次20bit的碰撞结果，若满足要求，则在序号中查找重复序号，若仍满足要求，对序号作order处理并输出
5. 计算难度值



数据组织如下：



数据buffer中存储的是经过一次桶排序的结果，桶排序个数为2^12个，每个桶（bucket）的大小为2倍的2^9（并包含一个32bit数，记录有效数据个数，可以单独使用一个数组），这是因为初始几次碰撞可能导致数据量增加，经过统计，一般不超过总数据量的3倍。

桶中的每个数据包括3个值，即value，上次碰撞（产生该值）的序号a和b，在初始的时候，a存储使用blake2b算法产生该值的序号，b存储原始数据值的前20bit，（注意，这样操作后，0~19bit被保留下来，不参与1~8次的处理，留到第9次进行一次性处理，则初始桶排序依据的值为第20~39bit）

Value值大小为180bit，collision a和b的大小为22bit（通常取32bit），方便起见总位数设定为256bit，部分位数是保留的。

一个buffer大小为2^12\*2^9\*32B\*3=192MB，需要2个buff，则数据空间尺寸为384MB。

Bucket中有效数据大小需要空间为2^12\*4B=16kB。

结果为512个32bit数值（实际占用20bit），通常结果个数为20个，则需用空间为40kB。

**1.2 一次碰撞主算法流程**

一次碰撞处理过程如下：

1. 将数据按2^9进行sub桶分类（外部是数据已经按2^12进行桶分类），桶分类保存时需要保存当前值的顺序号（从0开始）以及下20bit的值
2. 遍历sub桶分类结果，对结果大于2个的情况，作碰撞操作
3. 碰撞操作为22全遍历的配对，处理中以一个值为主碰撞值，其他的为从碰撞值
4. 将主碰撞值的(a,b)保存在old\_indices中，产生碰撞Xor值
5. 将Xor结果按外部数据桶分类关系进行保存，同时保存更新的(a,b)值（(a,b)值分为当前碰撞产生顺序计数号及主从碰撞步长）
6. 重复1）~6）完成外部全部桶的处理



数据处理要将一个bucket中的数据都读入，读入的数据按桶排序方式直接进入sub\_bucket，sub\_bucket大小为2^9\*17\*32B=272kB

在电脑软件中需要开的空间可能更少，仅需要保存下次用于xor的20bit即可，此外在代码中还产生了一个需要，用于标识当前数据的顺序，主要为了索引buffer内的数据，在本设计中由于将buffer中的数据全读取到内部，则不考虑反向索引，这样可以避免反复随机读写存储器

**1.3 第9次处理流程**

第9次处理过程如下：

处理1）~3）与一次碰撞处理相同

4）一旦成功碰撞则回溯主从碰撞值的全部索引号，并计算初始20bit的xor结果

5）若Xor结果为0，继续，否则返回3）

6）若索引号无重复，继续，否则返回3）

7）对序号进行order处理

8）保存结果，返回3）



**1.4 序号回溯技术**

为了节省空间，程序没有保存每次的索引序号，而是采用索引回溯的方式进行保存。

1）在产生数据时候，同时在每个数据下保存了(a,b)值，分别为产生该值的序号和前20bit数据

2）每一次处理时，对有碰撞的主值，将(a,b)值保存到old\_indices中，并产生一个顺序号（每次查找到碰撞都增加1），将新的顺序号和主从碰撞值步长记录到新的数据的(a,b)中

3）全部处理完成后，会得到一个关联表，根据任何一个(a,b)值，可以查找到上一层中的两个值，即由a和a+b定义的位置，该位置上仍然有(a,b)值，可以继续定位上一层，直到最初的列表，这样就能够得到全部的序列号，以及前20bit值。

下图给出了假设只有4次运算，indices存储阵列的情况



每级处理时，将当前一个sub\_bucket中的a，b值保存到indices阵列中，并顺序记录其在阵列中的位置（一行的序号），当进行22碰撞并产生新的xor值时，将响应碰撞的序号保存到新值下的a，b中，从而建立起索引关系



实际indices阵列一行的size不超过2^21+2^19个（不重复记录可以少些），indices矩阵行数为8行，则整个空间大小为8\*(2^21+2^19)\*8B=160MB

索引回读通过当前索引号取出下一次的索引号，总共512个，需要的内部存储空间为512\*8B=4kB（一个索引号包括a，b两个32bit值）

**1.5 Order处理**

算法要求产生的序号是有序的，因此需要重新排列数值，有序定义为将主从碰撞时，对于已有的序号全部拼接在一起（以前的数据是已经order的），然后比较大小，小的在前，大的在后。

算法的order是在最后处理的，order处理需要从第一次的2个比较开始排，第二次则用4个数排，直到第8次用256个数排，由于是当前的序号是已经order过的，因此每次比较仅需要比较第一个数值的大小，然后全部一起交换即可实现单次的order即可。



**1.6 需用存储空间统计**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 大小 | Ram or cache | 说明 |
| Buffer A | 192MB | Ram | 存储碰撞结果数据及两个索引序号 |
| Buffer B | 192MB | Ram | 存储碰撞结果数据及两个索引序号 |
| Buffer A Bucket Size | 16kB | Ram | 存储Buffer A中每个bucket中有效数据个数 |
| Buffer B Bucket Size | 16kB | Ram | 存储Buffer B中每个bucket中有效数据个数 |
| Solution | 40kB | Cache | 存储20个可能的解 |
| Sub\_bucket | 272KB | Cache | 存储当前处理需要的数据，包括192bit值和两个索引号 |
| Indices array | 160MB | Ram | 存储回溯索引号，共8层 |
| Temp\_indices | 4kB | Cache | 存储临时索引号，共512个，包括a，b两个值 |
|  |  |  |  |
| Total | 544MB+32KB | Ram |  |
|  | 316KB | Cache |  |

Ram存储空间可以控制在544MB（多余的32kB可以利用空余字节分摊，对FPGA来说可以放在cache中），cache数据是内部处理使用，软件一次性用完释放，对FPGA来说是需要开辟固定的空间。

为了节省存储空间，序号存储采用压缩方式

第1轮，存储8B（包括32b序号和32b数据的0~20bit）

第2~9轮存储4B（包括24b序号和8b距离差，实际可以调整一下）

这样序号矩阵大小为2^21\*8B+8\*(2^19+2^21)\*4B=88MB

总大小小于512MB

**2 blake2b算法**

该算法用于产生用于解决GBP的数据集，输入包括参数和消息，参数长度固定为64B（固定格式），消息长度不大于2^128B

每次运算都需要进行12轮迭代（迭代在mix器中进行），每次最多计算128B，依次推入计算块后，得到可以得到最终结果



其中mix需要多级运算，每级运算包括两次贯序的加和xor处理



Mix过程可以pipeline运算，每进入一个数据，需要经过12\*2\*2\*4=192clk，数据进入后，经过两个mix器，每个mix器运行12次，每轮需要运行4clk，为了提高效率，允许8个任务同时进入运算，则在控制上需要提供一个计数器及输入输出控制器

通用blake2算法需要考虑数据流大于128B的情况，在Equihash引用中，仅需要考虑32Bnonce+4B长度的情况，而且参数是固定的，可以提前算好，能够极大减少输入需求，当然也可以开放出来，设计中为了加速处理，采用并行总线，不考虑数据大于128B的情况

考虑到以下情况：

1. blake2算法核需要资源约11k一个
2. blake2算法运算轮数为12，即12个核周期才能完成一次运算，出现一个结果
3. 如果不采用pipeling方式工作，数据总线会存在加大空闲，为了填满空闲，需要较多的算法模块，资源消耗量基本相同，且控制较复杂
4. Equihash算法是经过设计的，输入数据长度不会超过128B，因此不存在核的迭代运算，仅就12轮核运算展开，就可以直接实现pipeline输出
5. Equihash算法不像ethash算法，数据集是一次性产生的，相反是每个nonce都要产生一次，那么数据集产生速度直接影响算法的处理周期，若采用pipeline方式产生数据，每5ns（200MHz）产生一个数据，共产生2^21个数据（即写入2^21次），则需要10ms时间，这决定了系统的最大处理能力为100次/s，假设有16个通道，则整个芯片的处理能力为1600次/s

出于以上考虑，拟采用pipeline方式工作，在总线分配上作文章，主要采用多个独立的总线通道，每个equihash对应一个固定通道，使用一个black2运算核，完成一个通道数据产生后，即开始下一个通道的数据产生，有数据的通道则进入GBP问题结算过程，期间也需要大量的数据访问

关于数据率的核算，采用pipeline方式下，1clk就能产生一个数据（50B），对应数据集中的2个数据，存储总线宽度为512b，则一次可以存2个数据，模块工作于200MHz，对应的存储器存储速率也要达到200MHz/512b，外部存储器带宽为64bit，工作速率要求为不低于1600MHz，考虑到数据采用DDR方式，实际工作速率大于800MHz即可。对应的数据存储次数为2^20次，则需要5ms时间完成数据集产生。

完成仿真，设计中，要求参数在计算完成前不能改变，实际中参数一般是不变的，否则内部需要一个fifo来缓存参数，消耗资源，目前没有设计，以后若资源多，且有通用需求，可以在内部增加一个，FIFO深度应不小于计算延迟

**3 FPGA实现**

**3.1 整体架构**

外部使用1块不小于544MB的RAM，用于存储buffer\_a，buffer\_b和indices矩阵，内部分别开辟一个Sub\_bucket，一个Solution空间，其他的空间可能用寄存器代替（实际可能也采用ram形式，例如Buffer A/B Bucket Size用ram存储，设计一套修正结构，能够事实修改里面的数值）。

内部尽量采用流水处理结构，完成一个处理流程，整个流程由状态机控制工作。

整个处理可以分为初始数据集产生（Blake2b），数据碰撞遍历处理，序号解压缩以及难度判断，为了设计方便，每个模块是相互独立的，由主控制器来控制各个状态的启动核结束，在主控制器中需要建立任务集，根据外部资源使用情况（主要是存储资源）来启动各个工作模块。

其中初始数据集产生模块仅实现1份，因为每个nonce计算都需要产生数据集，并且产生时间比较长，同时处理的时间比产生时间还长，过多数据集不利于系统效率提升，初步设想是数据集产生器每次产生一块数据，完成一块后，若由存储空闲，则再产生一块，这样可以充分利用该模块。

完成初始数据集产生后，控制器调用GBP解决器，该解决器被分为2部分，包括数据碰撞遍历核序号解压缩（之所以压缩是为了节省存储空间），GBP需要多次访问数据，可以认为需要9次读写数据，用时约为数据集产生时间的18~20倍（由访问次数估算）因此需要实现多个解决器并行工作，以达到多ram访问上的最大速率。

最后是难度判断，该判断仍然需要调用blake2b，不同于数据集产生，该运算数据长度超过了128B，需要多次调用，拟采用独立的模块来实现



考虑到Blake2b复用，在写入数据同时需要产生新的地址及数据整合，则进一步将整体框架更改为如下



该框架中，顶层使用一个任务状态机来控制内部模块，同时请求外部共用black2b模块产生数据集

内部分为3部分，包括外部Mem数据产生器（或者叫桶数据分配器），GBP解决器和序号解压器，

桶数据分配器包含两个桶数据分配模块，分别对应AB两份缓存，每个模块完成输入数据的桶排序及桶内数据计数，并产生外部存储需要的地址，两个桶数据分配模块通过一个路由器进行通道选择，能够确定当前的读取通道和写入通道，主要的写入通道有两个，分别来自black2b模块和GBP解决器，读取通道仅分配给解决器

GBP解决器包括一个桶数据分配模块和一个GBP状态机，每次读入一个桶内的数据，并进行进一步的桶排序（子桶），完成排序的数据存储到Ram中，控制模块判断每个子桶中的数据个数，然后产生序号并存储，最后将ram中的数据取出存储到外部ram中，控制状态机则需要产生各类读写信号，包括对桶数据分配器的配置信号，因为GBP需要多轮工作，外部两个buffer需要相互切换

序号解压器，通过读取序号并回溯序号来获得GBP问题的最终解，并将解推送回主控状态机中。

这样构架，只要延时对就可以了，另外外部Mem路由器需要设计一下，但是基本上读写分开，且不存在多个通道同时读写的情况，应该来说比较好做，通过配置进行数据通道选择

**3.2 桶排序分类计数器**

初始数据集产生需要完成2个功能，包括：

1. 使用blake2b产生产生hash数据，根据输入nonce+序号进行计算，每次产生50B（400b）数据，拆分为2个200b数据存储
2. 完成桶排序过程，即根据每个200b数据的20~39bit进行分类，并产生存储地址

设计中主要的问题是由于进行了桶排序，每个桶中可能存在多个数据，在产生地址时，需要根据已经存在的数据个数来修正地址，但是地址存储在ram中，需要一定时间读取并更新，这个时序需要考虑一下

数据存储时，每个数据规整为256b，使用192b存储180b数据，剩余位数存储32顺序号和0~19bit

Blake2b是pipeline设计的，可以实现1clk完成一次运算，但是这样一来，后边需要一次存2个数，存在一些问题（主要问题是桶排序中，桶内元素个数计算无法实现，使用reg太消耗资源，使用ram要用2个，且2个之间同步是个问题），因此在设计中计算一个间隔一个进行，输出数据则拆分为2次处理即可。

设计一个基于RAM的计数器，结构如下



由于需要的计数器较多，若使用寄存器可能需要很大的资源，故采用Ram来存储计数值，ram读取数据需要1clk时间，此外计数值增加也需要1clk时间，那么连续两次对同一个计数器进行增量触发，可能导致最终结果不正确，维持采用一个带有延迟结构的逻辑来进行计数处理，输入前3次的地址（分别为ABC）若A=B=C则增加3，若A=B或A=C或B=C增加2，其他情况增加1这样就能保证Ram中的值是最新的，且计数出来的结果也不会有错误



此外还要增加一个读取功能和一个初始化功能（计数器是单向增加的，若需要加减功能，可以额外增加一个控制线，经过1clk延迟到达逻辑中进行判断即可，但是逻辑中同样需要记录n-1，n-2次的状态，以实现真是的加减处理，暂时不做，等有需求了再扩展）

初始数据集产生模块只有一个，但是计数器是跟随模块数的，按存储模块的划分，每个存储模块可以对应一个计数器，在架构的时候Blake2b模块通过路由来选择初始化（Init）及增加计数器的值（Inc）

存储地址由3部分组成，包括每个模块的基础地址，每个桶的偏移地址和桶内数据位置偏移地址，其中基础地址是固定的，每个模块（线程）分配一个，桶偏移地址由Blake2b计算结果中20bit的前12bit决定，并乘上桶地址跨度，桶内数据偏移地址由计数器给出，每完成一次blake2b计算，进行桶地址计算，并增加对应桶内的计数值。

根据新的架构设计，需要一个通用的桶排序分类计数器，以已经设计的基于ram的计数器为基础进行包装，封装成适合桶排序地址数据产生的模块，FPGA内部结构如下



数据输入，分离出需要桶排序的12bit（可配置），进入计数器产生桶内序号，此外该值还需要乘以桶地址基数，最后与buff基地址相加，产生最终用于存储的地址。

数据路由通过配置通道关系来实现通道控制，每个控制器有2个输入和2个输出，对应有2种组合，即AABB和ABBA，同时设置有2个输入控制，包括主控配置和从控制配置，若两者一起到来，以主控配置为优先





关于存储通道顺序，见下表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | 控制信号 | IncSel | GetSel | ChSel |
|  |  | 控制信号说明 | 写入计数器选择  0：A端口写入到A计数器中  B端口写入到B计数器中  1：B端口写入到A计数器中  A端口写入到B计数器中 | 读取计数器选择  0： A计数器读取到A端口  B计数器读取到B端口  1：A计数器读取到B端口  B计数器读取到A端口 | 数据输出通道选择  0：A计数器通道输出到o端口  1：B计数器通道输出到o端口 |
| DBG | 接入A端口 |  | 0（A->Am） | x | 0（Am->o） |
| GBP | 接入B端口 | Round 0 | 0（B->Bm） | 1（Am->B） | 1（Bm->o） |
|  |  | Round 1 | 1（B->Am） | 0（Bm->B） | 0（Am->o） |
|  |  | Round 2 | 0（B->Bm） | 1（Am->B） | 1（Bm->o） |
|  |  | Round 3 | 1（B->Am） | 0（Bm->B） | 0（Am->o） |
|  |  | Round 4 | 0（B->Bm） | 1（Am->B） | 1（Bm->o） |
|  |  | Round 5 | 1（B->Am） | 0（Bm->B） | 0（Am->o） |
|  |  | Round 6 | 0（B->Bm） | 1（Am->B） | 1（Bm->o） |
|  |  | Round 7 | 1（B->Am） | 0（Bm->B） | 0（Am->o） |
|  |  | Round 8 | 0（B->Bm） | 1（Am->B） | 1（Bm->o） |
|  |  | Round 9 | 1（B->Am） | 0（Bm->B） | 0（Am->o） |

**3.3 GBP解决器**

GBP解决器本质上是一个过程，可以用一个状态机来实现，器包括如下过程：

1. 清空子桶排序器，配置主桶排序分类计数器，并读取一个桶内的数据个数
2. 从存储器总将桶内的数据读取出来（读取出来的同时会执行子桶排序，不需要状态机管）
3. 完成数据读取后，从子桶排序器总读取一个桶内的数据个数
4. 若数据个数小于等于1，则读取下一个，若大于1，则从内部存储器中读取该子桶内的标记号，并存储到存储器中，同时更新标记号
5. 然后从内部存储器中依次获取数据，并执行执行处理后（不用状态机管），输出到主桶排序器中
6. 完成全部子桶排序
7. 完成10桶排序过程，每次对应的碰撞字段不同

处理过程中，需要如下计数器：

1. 桶排序过程次数计数器，同时需要给出数据块选择值0~9
2. Buffer地址选择，需要在AB两个buffer之间切换，此外控制外部路由器的配置（以后还要增加对总线分配的控制）
3. 数据个顺序数计数器，其需要作为更新数据索引的值，在一次循环中一直计数
4. 一个子桶内数据个数计数，该计数是个双重计数，在状态机上有两层循环，用于取出主碰撞数和从碰撞数，并输出从碰撞数相对主碰撞数的步长



梳理状态机需要的输入输出，及产生的计数值关系

1）初始配置

初始配置包括配置：路由，buff基础地址，清除sub\_Bucket计数器等

输出IncSet和GetSet，输出信号根据当前工作轮数来确定，IncSet偶数轮采用A->A模式，奇数轮采用A->B模式（GBP模块是接到B通道的，初始数据是通过A通道存储的），GetSet则相反

输出AB\_BuffSel，用于选择哪个buff的数据，选择在外部进行

输出subB\_Init/B\_Init，并等待subB\_Rdy/B\_Rdy反馈，表明sub\_Bucket/bucket中的计数器被清零

2）读取一个Bucket中的数据个数

输出Get和GetIdx，其中GetIdx需要一个计数器q，遍历全部的Bucket（0~2^12-1），返回当前Bucket中的个数Bn(q)，当个数大于1时继续执行，否则重新取下一个数

3）读取ram中的数

输出Addr，该地址需要修正，Addr+AB\_Bucket\*q+AB\_Buff

Ram读取个数为当前Bucket中的个数Bn(q)使用一个计数器进行计数

4）读取sub\_Bucket中的数据个数

输出sub\_Get和sub\_GetIdx，其中sub\_GetIdx需要一个计数器p，遍历全部的sub\_Bucket（0~2^9-1），返回当前sub\_Bucket中个数sBn(p)，当个数大于1时继续执行，否则重新取下一个数

5）读取sub\_Bucket(p)中的索引，保存到外部并更

读取索引操作，输出sub\_Addr，该地址需要修正，sub\_Addr+AB\_sub\_Bucket\*p，其读出的值将被送到外部存储器中进行存储（32bit）

外部存储地址，输出Idx\_Addr，该地址需要修正，r\*AB\_IdxArr+AB\_Accm+Idx\_Addr，其中AB\_Accm是一个计数值，保留了前面已经加过的数的个数

更改操作紧跟在读取操作后面，输出一个写信号，并且将AB\_Accm+Idx\_Addr写入进去，当然也可以直接累加AB\_Accm，并写入

6）读取sub\_Bucket(p)中的索引和数，并保存

该读取分为两步执行，包括读取主数，和从数，在碰撞中主数读取一次，从数读取多次

主数输出sub\_mAddr，该地址需要修正，sub\_mAddr+AB\_sub\_Bucket\*p

从数输出sub\_mAddr+j，该地址需要修正，sub\_mAddr+AB\_sub\_Bucket\*p+j，该j即为主从数的步长，从1开始计

主从数碰撞过后的结果存储在外部，该地址不再由本状态机产生

|  | | 状态1 | 状态2 | 状态3 | 状态4 | 状态5 |
| --- | --- | --- | --- | --- | --- | --- |
| 前向跳转路径 | | 状态2 | 状态3 | 状态4 | 状态5 |  |
| 后向跳转路径 | |  | 状态1 | 状态2 |  | 状态3 |
| 循环次数 | | 10->r | 2^12->q | 2^8->p | sBn(p)->j | sBn(p)->j |
| 处理内容 | | 初始配置 | 取Bn(q)->i  读取数据 | 取sBn(p)  判断是否继续 | 读取Idx，存储并更改  累加Accm | 读取数据并存储  两重循环  j = 0~ sBn(p)-2  k = j+1 ~ sBn(p)-1  输出步长 |
| Bucket | | S\_Config IncSet GetSet  Init Rdy  chunkSel  AB\_Buff\_Wr | Get GetIdx=q  Cnt = Bn(q) |  |  | Inc |
| Sub\_Bucket | | Init Rdy  AB\_Buff\_Rd  chunkSel |  | Get GetIdx=p  Cnt = sBn(p) |  |  |
| Mem | Rd |  | A: i+AB\_Bucket\*q+AB\_Buff |  |  |  |
| Wr |  |  |  | A: AB\_IdxArr+r\*AB\_IDArr1+(Accm+j) |  |
| Ram | Rd |  |  |  | A: j+AB\_SubBucket\*p | A: j/k+AB\_SubBucket\*p |
| Wr |  |  |  | A: j+ AB\_SubBucket\*p  D: Accm+j |  |
| 设置参数 | | r |  | p sBn(p)  reset Accm | Accm |  |

处理过程分为5个独立的状态机工作，通过控制信号来协调（不在乎短暂的控制空闲）

外部制作一个地址运算器，完成一次乘一次3加运算，每个状态机仅需要输出基本参数即可，计算器延迟2clk，状态机设计中需要注意，读写信号等会在外部进行延迟，计算时全部使用32bit运算，乘法结果为64bit，然后裁剪为32bit输出（问题是这样做会不会导致多使用乘法器，这个需要考虑一下）



~~统一参数传递接口，使用4个32bit数，编译器自动会删除不用的参数传递~~，不用了传递参数了，各个状态机外部参数保持即可，状态机之间使用反馈信号控制，不存在同时工作的情况

注意：状态3可能需要重新设计，由于读取cnt存在延迟，理论上来说，需要读取2^20\*10次，若每次延迟4clk，则全部读取一遍，耗时就已经达到200ms（5ns时钟），其用时比读取数据还要多，为了加快速度，可以将状态3拆分为2个状态，一个状态启动后不停的读取cnt，若cnt大于1，则准备号需要运算的参数，等待另外一个状态机获取，一旦获取后，再开始查找下一次（其中应当记录当前已经读到的位置，下一次从该位置继续），另外一个状态机则直接敏感数据准备好信号，可以实现每个clk推出一个任务，这样设计，可以将延迟认为是1clk，对应的耗时可以控制再50ms左右，与数据读取时间相当（数据读取2^21个，读取4次，对应的耗时约40ms）

这样考虑后，后级可能需要做紧凑设计，以尽量减少工作时间，该系统由于无法进行超前运算，因此减少两个状态机之间的跳转控制间隔，能够有效的降低时间



考虑到处理速度的问题，若总线带宽足够，可以考虑多个子桶同时处理，以提高数据处理速率，对于已有的设计可以这样更改：

1. 状态1的 round计数功能不改变
2. 状态2读取子桶数据的功能保持，增加子任务分配器，向空闲子桶处理器发送任务，并收集任务结束信号，根据子任务配置读取通道，这部分可以单独做一个模块
3. 状态3~5功能不变，其集合称为一个可重复的状态过程
4. 对于mBucket，需要对应n份（与子任务相同），每份中仅记录了部分的桶内存储个数，那么在状态2中需要读取全部值然后相加得到桶内的真实数据个数
5. 状态2的CacheSel信号更改到状态3中产生，状态1中对sBucket初始化过程放在状态3中执行，状态3初始化需要执行一次，然后等待任务到来，任务结束前也需要处理一次，以准备好下次任务到来

这样整个架构需要更改如下

状态1+2及状态3+4+5组合更改如下







更改中有一个问题未解决，每个线程在新Idx计数中，起始位置是接续的，因此需要一个统一的Idx计数器，接受来自每个线程的Idx地址申请（其同时提供需要多少个地址，用于计数器计数更新），每个线程根据地址计数器给出的值进行Idx计数及存储地址产生（这个以后改）（已更改）

在最后一次collision过程中，不再需要存储数据，但是需要额外单独存储一次序号

注意：实际只使用前180bit工作，即9轮，数据产生为200bit，一次产生400bit分为2个来存储

Order处理也分为4个状态机，集中在一个状态中访问存储器

1. 控制主循环，通知后级读取第几个数据
2. 负责访问存储器过程（读取的数据解包在存储过程中解码并存储，一次读取1个数存储2个数，存储器配置为输入输出数据宽度不同，输入64bit输出32bit，对应的输入2个数据，输出1个数据）
3. 负责查找是否有重复数，允许提前结束过程（重复读取数据，每次读取完成后，判断判断结果，若为1，这直接abort）
4. 负责数据组合的比较，并交换数据位置（读取两个数据进行比较并交换存储）



Idx解压缩需要读取10次

完成原型设计，存在的主要问题是：

1. 由于stp3大量读取sBucket中的cnt值，每次读取都需要等待，导致大量时间冗余
2. Stp4读取和修改索引值需要两次读写耗费时间
3. Stp4和5功能中写索引和写数据过程是可以重叠的，独立工作会导致使用时间增加

基于以上原因，需要对子线程做重大修改，主要改进如下：

1. stp3利用FIFO来缓存sBucket中的cnt值，以任务形式推动后级工作，这样就可以连续读取cnt值，减少等待周期（FIFO用almost full来指示是否慢，因为sBucket输出有流水延迟）
2. 合并stp4和5
3. 增加一个内部idx存储器，用于存放更新索引号，序号从0开始
4. 在sBucket数据输入同时，产生该子桶内的更新索引好，stp4不再单独产生新索引号
5. Stp4和5对数据即索引的存储同时进行（除了最后一次，两次索引输出需要顺序进行），这样合并的状态机可以也是两个，不过两个是同时执行的
6. 原先在Stp4中执行的序号申请，移动到stp2中进行，每次获取mBn后，申请一段序号，stp4就以此为基础进行累加工作即可，序号累加器存储在stp2中即可

另外stp1也可以按FIFO流失模式设计，提高前级工作效率（待设计）

解序号也序号做多线程处理（待设计，输出采用申请模式，每次仅允许一个通道输出即可，毕竟输出结果是少数）



**3.4 数据集产生**

一次产生2个数据，控制器每2clk变化一次计数值，并输出2clk的Trg信号（该信号经过延迟后作为bucket分类器的Inc使用）

外部可直接接入Bucket中，控制信号需要经过延迟，输出数据也需要经过延迟



测试改动：

1）Equihash\_DBG\_Wrapper

component Equihash\_DBG\_Ctrl

generic(

Num\_Idx : Natural := 10000 -- (only for test)2\*\*(gcst\_W\_Chunk) -- 2^20

);

减少产生数据个数

2）Equihash\_GBP\_CllsSDisp

if(sgn\_q = 10-1)then -- (only for test)last bucket is in process

减少mbucket访问次数

3）Equihash\_PoW\_Wrapper

use ieee.math\_real.all; -- only for test

优化设计更改需求：

1. Blake2b支持使能控制，由写存储器valid来控制计算过程，一旦对应的存储器valid无效，下一拍停止运算，同时DBG也停止工作（需要包括所有的控制延迟信号，sel，trg，idx等）ok
2. DBG部分独立拆解出来，并支持多线程，需要增加产生数据起始序号及个数输入
3. 将DBG结束信号延迟整合到DBG模块中，方便拆解ok
4. GBP求解器支持多个线程，增加外部存储器valid控制ok

|  |  |  |  |
| --- | --- | --- | --- |
| 接口组 | 功能 | 源头 | 说明 |
| Mem\_p1 | 读取数据 | MThread->GBP\_stp2(ok) |  |
| Mem\_p2 | 写入index | SThread->GBP\_stp4(ok) | Stop state machine |
| Mem\_p3 | 写入数据 | SThread->GBP\_stp4(ok)  DBG(ok) | mBucket\_Inc |
| Mem\_p4 | 读取index | UnC->Stp2(ok) |  |

**3.5 v2版本设计架构**

经过大量设计更改，GBP设计结构包括一个主控模块+多个子线程模块+多个mBucket模块（该模块为暂时的，实际中需要减少到1个模块，利用多通道网络进行数据路由后，进入mBucket中，以减少计数器个数），经过仿真结果正确。



主控模块：负责GBP任务循环，多线程分配等



子线程结构：完成数据获取，数据分类，数据碰撞，序号更新及数据输出



为了适应外部使用，还需要作如下更改：

1. Blake2b支持使能控制，由写存储器valid来控制计算过程，一旦对应的存储器valid无效，下一拍停止运算，同时DBG也停止工作（需要包括所有的控制延迟信号，sel，trg，idx等）ok（已仿真）
2. 增加外部存储器valid控制（仅更改状态机，不考虑流水线），更改内容包括ok

* sThread -> stp4 -> mB\_valid\Mem\_Idx\_valid ok
* sThread -> stp2 -> Mem\_D\_Valid ok
* Unc -> stp2 -> Mem\_Valid ok

1. 地址偏移都改为输入可配ok
2. 线程分配模块增加一个msk用于屏蔽不用线程（实际中进程数量可能达到128个之多）ok
3. 增加输入sBucket数据拆分模块（一个FIFO+一个拆分器），配合数据拼接情况，此外在rd信号输出的时候也要增加一个读取拼接器，注意数据可能不足，需要提前配置msk，msk需要自动计算（作为额外部件添加）