多线程设计思路

Equihash整个过程分为3部分，包括：

1）数据集产生（DBG）

2）GBP算法部分1——数据碰撞

3）GBP算法部分3——索引解析

由于算法在数据碰撞时需要大量随机访问存储器，且存储器带宽较宽，为了将带宽占满，需要采用多线程的方式工作，根据算法设计，Equihash过程的3个部分都可以进行多线程设计

存储器接口模式说明：存储器以512MB为单位，每个单位外部对应8组接口，一个进程要求占满全部带宽，需要采用多线程方式工作，包括读和写操作。

1 DBG

DBG主要使用Blake2b算法产生随机数据集，blake2b每次产生2个数（各200bit），数据存储时分2次存，每次存储包括200bit数据和56bit参数（共256bit）

DBG产生根据一个nounce及顺序序列号产生，序列号从0到2^20-1

由于以上输入方式，DBG可以用多线程工作，每个线程执行一段规定的序列号，模块可以顺序输出blake2b结果，并产生存储地址。

这里需要注意的是数据在存储的时候根据碰撞字节，进行一次桶排序，排序结果即为当前数据存储的结果，同时在模块中记录当前桶中的数据个数，在多线程设计中，这个需要额外注意，因为多线程下，每个线程都可能产生数据个数，最终使用的时候需要将各个线程的结果加和才能使用，地址产生模块是独立的，不包含在DBG中。

2 GBP算法部分1

GBP算法部分1主要解决数据碰撞问题，主要完成任务包括：

1. 从存储器中获取数据
2. 将数据按碰撞字节进行桶排序，放到sub-Bucket中
3. 判断每个sub-bucket中的数据存储old-idx，并执行xor操作产生新的数据及new-idx
4. 输出新数据到存储器端口

由于算法特性可以进行多线程操作，数据在初始存储的时候已经进行了桶排序，即数据按碰撞字节存放在m-bucket中，GBP算法部分1就是将每个m-bucket中的数据取出执行，一个线程一次可以完成一个桶内数据的碰撞，则多个线程可以同时对多个桶内的数据进行碰撞。

碰撞完成后，数据经过桶排序产生地址并存储到外部存储器中。

此外模块会额外产生序号的存储请求，针对由碰撞的情况将old-idx存储到一个二维数组中。

这里需要注意的是，m-bucket排序模块包含在该部分中，以方便设计和多线程化

3 GBP算法部分2

GBP算法部分2主要根据idx数组中的值，产生最终的GBP问题解，主要步骤包括：

1. 获取初始idx
2. 回溯idx，获取全部512个碰撞idx
3. 判断idx中是否存在重复
4. 对idx进行排序操作
5. 顺序输出idx

该过程是一个整体，不再拆分，可以例化多个该模块，进行多个idx同步解析，提高运算发速度。

每个多线程模块单独设计并提交，线程调度及需要配置的工作产生，根据当前过程执行，具体配置按模块设计，GBP问题的9次碰撞过程也由控制流程执行。

每个线程对外存储器访问以直接输出读写控制信号即地址的方式进行，不关心具体的总线仲裁（意味着外部需要用FIFO来承接读写请求），在读写方面模块自身有读写请求valid信号来控制是否进行读写。

以上3个线程模块统一设计统一部署，部署的可能个数为：DBG线程8个（多进程共用），GBPp1线程8个（每个进程开辟8个独立线程，包含地址分配器），GBPp2模块1个（一个进程提供一个），模块的互联，总线仲裁等问题由顶层设计解决。

整体架构如下图所示（仅仅绘制数据流）



DBG模块设计

DBG模块使用多个线程产生数据，数据产生个数是固定的2^21个，产生次数为2^20次，若采用8个独立线程，可以分配每个线程产生2^18个数据（运算2^17次），可以由主控系统配置运算起始位置（nSt）和次数（nNum），此外还要输入nounce等参数

DBG模块输出直接通过Mix网络进入GBPp1模块，进行桶排序，并生成需要存储的地址，同时进行桶内数据量计数。

DBG需要接收来自外部的valid信号，以控制数据流过程，即运算模块能使用使能信号停止工作。

GBPp1模块

该模块设计多个独立线程工作，每个线程包括以下子模块：

1. 主桶排序及地址产生模块
2. 数据读取模块（GBP\_stp2）
3. GBP碰撞处理模块（GBP\_stp3/4）
4. 子桶排序及内部缓存

由于采用多线程工作，GBPp1不再执行GBP问题的主循环（待定），需要外部进行控制，一次GBP过程是对一个主桶中的所有数据进行碰撞，完成碰撞后会反馈结束标志。



设计注意点：

1. GBPp1的stp2只输出读取个数，线程模块内部不计数读取请求个数或反馈个数，再线程外部设计2个计数器，分别累加读请求个数和反馈个数，多个线程的2个计数器都达到总个数后给出标志，所有线程统一进入stp3（见控制器）
2. GBPp1的stp2配置一个FIFO存储需要读取的mBucket中数据读取个数，外部设置一个控制器，统一从fifo中读数，并启动多线程，然后收集多线程的结束信号，当所有线程都结束后，再读取下一个数（见控制器）
3. GBPp1的sBucket解析地址仅解析一部分，需要外部配置
4. GBPp1的mBucket内部存储器分解为多个存储器，以加快清空的速度，否则太慢了
5. GBPp1的stp4最后一轮每个线程都输出到一个fifo中（结果较少，可以使用很小的fifo，深度32足够了，内部有fifo-full使能控制），再通过外部请求来统一输出到外部FIFO中（？？）
6. GBPp1中每个Mem\_idx的AB，需要考虑每个线程由多少个数，第一个线程从0开始，第二个线程需要加上第一个线程接收的数据量a1，以后依次类推，第i个线程需要将前i-1个线程接收的数据量都加前来，作为AB的修正值，AB即从该位置开始存储，这个有点复杂，而且无法再单元模块中完成，读取完数后，需要几个clk来保证AB的正确（见控制器）
7. 主控stp1获取mbucket中的个数，需要将每个线程中的mBucket的个数加起来，然后再分配
8. 主控stp1输出个数后，不启动线程，交由另外一个线程启动，主控根据FIFO是否全空及线程是否全结束来判断当前round是否结束
9. mBucket和sBucket的数据宽度仅需要5/12bit（对应17和3\*2^9个数）
10. mBucket仍然放在线程外部，这样原来主模块更改较少

综合以上注意点，在多线程外部需要额外增加如下控制器：

1. 数据读取结束控制器：用于GBPp1的stp2读取数据，每个线程配置两个计数器，每读一次计数器A+1，每反馈一次计数器B+1，控制器将全部计数值都加起来，若等于mBn则给出读写结束信号（需要延迟nclk反馈给stp2，因为外部计算Mem\_Idx基地址需要时间）
2. 线程启动控制器：用于启动GBPp1的stp2，即启动线程工作，其收集全部线程中的读取数据参数FIFO的emp信号，若全有，则说明至少有一个任务，那么给出FIFO读信号，并启动stp2，然后收集多线程的结束信号，全部接收，再进行下一次任务
3. Mem\_Idx基地址产生器：用于GBPp1的stp4计算Mem\_Idx地址产生运算，根据每个模块的输出地址来顺序累加获得各个模块需要的结果，则要求stp2结束信号延迟nclk（n为线程数）
4. Stp1多线程主控器：主控器任务为产生每个线程需要工作的参数，包括round，LastRound等，控制完成9轮运算，每轮运算都需要读取mBucket的个数，并推送到每个线程的FIFO中，同时监控FIFO集各个线程的状态，确定当前round是否完成
5. mBucket个数加法器：将多线程的mbucket个数累加后输出给主控器

现有代码更待记录：

1. Equihash\_GBP\_CllsSThread增加stp2部分及相应的计数等模块
2. Equihash\_GBP\_CllsStp2改为直接获得mBucket值，并增加控制线
3. Equihash\_GBP\_CllsStp1保持不动，其主要用于切换mBucket，并计数round
4. Equihash\_GBP\_CllsSDisp需要根本性改动，增加读mBucket值的过程，并判断round是否结束（每个线程中mBucket缓存FIFO需要多流几个空间，以方便主控设计）
5. Equihash\_GBP\_CllsStp4看看最后那部分怎么改吧，需要判断FIFO控制，这个有点问题

考虑到ASIC资源及实现难度等问题，对于以上设计作图下变更：

1. 仍然将GBPp1的stp2合并到子线程中，但是一个线程处理一个mBucket，延续以前的多线程设计方法，主线程仅负责将每个mbucket中的数推送到线程中即可，推送内容包括mBucket序号（q），当前mBucket中数据量，以及Idx存储起始位置，以原来的模块为主，将stp2拆分为2个，一个放到线程中即可
2. mBucket只使用一个，需要增加一个数据路由网络来对数据进行早期分类（路由网络已经有了）

20180913完成设计及仿真结果与1通道情况一致，更改后的模块如下：

线程控制模块



子线程模块



模块组合

