Ethash 算法部分设计说明

**0 概述**

Ethash从属于Ethereum的PoW部分，遵循Ethereum1.0版本，算法部分包括2部分，即DAG数据产生和hashimoto循环

DAG数据产生包括以下部分：

1）根据交易周期等信息，计算cache及DAG数据集大小

2）根据当前seed计算规定大小的cache数据

3）根据cache数据计算规定大小的DAG数据集

Hashimoto循环时验证nonce的主要处理，利用header和假定的nonce并利用DAG数据集来确定一个计算结果，其利用了加密学的一些原理（FPGA实现不关心）。

算法实现需要一些固定参数如下：

WORD\_BYTES = 4 # bytes in word

DATASET\_BYTES\_INIT = 2\*\*30 # bytes in dataset at genesis

DATASET\_BYTES\_GROWTH = 2\*\*23 # dataset growth per epoch

CACHE\_BYTES\_INIT = 2\*\*24 # bytes in cache at genesis

CACHE\_BYTES\_GROWTH = 2\*\*17 # cache growth per epoch

CACHE\_MULTIPLIER=1024 # Size of the DAG relative to the cache

EPOCH\_LENGTH = 30000 # blocks per epoch

MIX\_BYTES = 128 # width of mix

HASH\_BYTES = 64 # hash length in bytes

DATASET\_PARENTS = 256 # number of parents of each dataset element

CACHE\_ROUNDS = 3 # number of rounds in cache production

ACCESSES = 64 # number of accesses in hashimoto loop

**1 DAG数据产生**

**1.1 cache及DAG数据集大小**

数据集大小根据当前数据集尺寸，增长量及历史更新次数来综合确定，运算过程中有判断质数的操作，因此不适合直接用FPGA计算，通常采用查表方式实现即计算足够多的数据表集大小，存储在存储器中，根据当前历史更新次数来查询。

一般来说该运算时间较长，可以一次离线运算完成后存储，数据应当存储在非易失存储器中。

def get\_cache\_size(block\_number):

sz = CACHE\_BYTES\_INIT + CACHE\_BYTES\_GROWTH \* (block\_number // EPOCH\_LENGTH)

sz -= HASH\_BYTES

while not isprime(sz / HASH\_BYTES):

sz -= 2 \* HASH\_BYTES

return sz

def get\_full\_size(block\_number):

sz = DATASET\_BYTES\_INIT + DATASET\_BYTES\_GROWTH \* (block\_number // EPOCH\_LENGTH)

sz -= MIX\_BYTES

while not isprime(sz / MIX\_BYTES):

sz -= 2 \* MIX\_BYTES

return sz

**1.2 cache数据计算**

Cache数据计算利用已知的seed进行，包括2个相对独立的部分：

1）根据当前seed贯序计算初始cache数据o，充满规定的cache空间（大小由外部输入）

2）对初始数据进行randmemohash操作，该操作利用当前cache数据（oi）计算访问序列号，并取出该序列号中的数据执行hash操作，作为oi的新数据，该操作需要执行3次



**1.3 DAG数据集计算**

DAG数据集计算以cache数据为基数产生，每个数据计算分为3个部分，数据位置序号是其唯一输入：

1）从cache中获取规定位置的值，并计算初始hash

2）计算256次随机位置，取cache值，并进行fnv操作获得mix结果

3）对mix结果作hash作为当前DAG的数据



**1.4 FPGA实现**

**1.4.1 cache过程**

过程1与hashimoto的过程1基本相似，使用一个hash执行cache\_size次运算，运算结果存储到外母Mem中，其中，写入数据由Mux锁存，写请求，写地址有DFC产生，DFC在收到ack后执行下一个循环，直到全部cache数据产生完毕



Proc2仍然使用Hash512，为了节省资源（时序上是串行处理的），将两部分合并位一个部分执行，并公用一个状态机，处理过程如下：

1. st到来后进入P1处理过程，该过程中每完成依次hash后，存储一个数据，输出计数值i最为地址，响应Ack\_R信号，共进行cache\_size次处理和存储
2. 完成cache初始化后，进入P2处理过程，该过程基本过程为
3. 产生计数值i
4. 获取对应cache值，输出i作为地址，响应Ack\_A1（输入数据经过mod处理后形成新的地址A1）
5. 输出A1作为地址，响应Ack\_O1（输入数据被锁存在Mux输出端）
6. 输出A2作为地址，响应Ack\_O2（输入数据经过Xor后锁存在Mux输出端）
7. 发出Hash\_st信号，响应Hash\_ed，完成hash处理后，请求数据写入cache中，地址为i
8. 重复3~7步骤直到i计数达到cache\_size
9. 重复P2 3次

注：对Mem有读写2中情况，其ack是一个，但是req需要分为读req和写req

注：将sel减化为延迟线，在输入端使用逻辑与进行选择

注：(n-1+i)%n等效于i=0时取n-1，其他时取i-1

注：考虑到连续两次o1和o2读取是紧接着的，但是在思绪上要求隔开一个clk，因此MUX和状态机触发信号虽然是同一个，但是两者需要差1clk

同样cache数据存储后，进行下一次读取两者间需要延迟一个clk，触发信号使用同一个但差1clk



**1.4.2 DAG过程**

DAG过程同样需要使用到Hash运算，并且要多次读取地址，在结构上与前面设计相似，使用一个Mux锁存器存储数据，通过控制流完成数据处理流程控制。



**2 Hashimoto循环**

**2.1 算法流程分析**

Hashimoto循环输入仅有header（32B）和nonce（8B），通过DAG数据集来获取最终的计算结果，整个过程分为3部分：h

1）计算header+nonce的hash值

2）计算DAG数据位置，并查表获得DAG值，循环64次

3）对结果进行压缩处理获得最终值



**2.2 FPGA实现**

**2.2.1 proc1/3设计**

在FPGA时间时，考虑到需要复用Hash单元，因此将proc1和proc2过程合并在一个模块中实现，由于部分数据需要延迟到后级使用，因此需要增加一些所存单元。

控制过程如下：

1. St信号到来后，将Head+nonce锁存
2. 数据流控制器（DFC）接收到St信号后，通知Hash工作（之前已经将Mux切换到头部输入状态，并且HASH工作参数已经配置到端口）
3. DFC等待Hash完成处理，hash模块会产生hash\_ed信号，该信号会将hash结果锁存
4. DFC产生控制信号Proc2\_st，通知Proc2工作（至此，第一个处理流程完毕），此外外部参数需要切换MUX控制及配置Hash工作参数
5. 但Proc2完成处理后，会给出Proc2\_ed信号，该信号经过延迟，将Proc2的处理结果压缩成32B，锁存到寄存器中，同时通知DFC
6. DFC接收到信号后，通知Hash工作（参数已经配置）
7. DFC等待Hash完成处理，hash模块会产生hash\_ed信号，该信号会将hash结果锁存
8. DFC产生ed信号，表示整个处理流程完成。



为了节省寄存器，将输入锁存和Mux合并为一个clk完成，整体架构变为如下（主要寄存器集中在FNV运算单元中）



**2.2.2 proc2设计**

处理过程2架构如下，在产生地址但基本是流水的，产生地址后，处理会中断，等待数据到来后继续，控制信号经过延迟触发DFC，产生必参数变化。



**3 Hash运算**

**3.1 结构及输入输出设计**

在Ethash中多次使用到Hash运算，包括hash256和hash512，hash运算采用kecck\_f[1600]方法(注：内部运算12+2\*6=24次，2^6=64，1600=5\*5\*64)。根据Ethash算法，其使用hash操作是经过设计的，每次输入的数据都为一段，即Keccak的海绵吸收结构（sponge）只有1级且输出结构（squeaeze）不需要再次运算映射函数f，（这里也可以看到为什么输入为96B时使用Hash256，若使用Hash512，就需要2级吸收结构）



初始时r，c为全0，因此xor可以直接简化为原始数据，不足位数padding即可（r部分按要求填充，c部分保持全0），在算法中，输入hash的长度是固定的（hash512输入长度为64B，hash256输入长度为96B），输出长度也是固定的（64B）因此整个Keccak算法模块可以极大简化。

对hash输入若长度不足r的长度，需要进行填充（padding），填充值为10\*1（即1后面多个0，也可以没有，最后是一个1），对于Ethash来说，其采用推荐的安全等级，即c的长度为hash结果长度的2倍，则有

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hash类型 | Hash码长度 | c长度 | r长度 | 输入长度 | 需要填充长度 |
| Hash256 | 256b | 512b | 1088b（136B） | 768b（96B） | 320b |
| Hash512 | 512b | 1024b | 576b（82B） | 512b（64B） | 64b |

由此可以看到，除了输入填充长度不同外，其他运算是一致的，则2种Hash可以一个模块实现，通过配置来区分。

**3.2 映射运算**

Keccak运算函数根据选择的1600长度，需要执行固定的24次运算，每次运算包括：

1）θ操作

该操作分为2步执行：

1）沿x维数据按位xor

b(x) = a(x,0) ^ a(x,1) ^ a(x,2) ^ a(x,3)

2）a矩阵数据交织

a(x,y) = a(x,y) ^ b(x-1) ^ rol(b(x+1),1), y=0~4

该操作意味着当前a(x,y)的值被x相邻位置上的b改变，且b(x+1)的值不在同一行（即bit位循环移动一次）

2）ρ操作

该操作仅对二位矩阵中的每一个元素进行循环移位操作（64bit），根据不同的x，y值通过查表获得移位数

a(x,y) = rol(a(x,y), ρ(x,y))

3）π操作

该操作是对数据在x-y二维上进行旋转（45deg），实际中旋转位置是固定的，可以通过查表获得数据交换位置

a(x,y) = a(π(x,y))

4）χ操作

该操作是一个逻辑操作，按x维进行，具体如下

a(x,y) = a(x,y) ^ (not a(x+1,y) and a(x+2)), y=0~4

5）ι操作

该操作仅对a(0,0)执行，对其乘上一个确定的值，可通过查表获得，不同运算次数值不同

a(0,0) = a(0,0) ^ RC(i), i=0~23

根据上述描述，ρ操作和π操作是不需要运算的，在FPGA实现时仅仅交换连线位置，θ、χ、ι操作仅需要进行逻辑运算，共4次运算，且每个bit的扇出不大于3次，这个逻辑级约为4级（实际上仅a(0,0)位置存在4级逻辑级），因此一般可以在一个clk中完成运算。



为了减少两级之间的逻辑层次，FPGA设计中，使用了3级寄存：第一级完成数据输入切换及累加矩阵生成θ1，第二级完成bit交织θ2，第三级完成χ和ι操作，使用控制器按节拍控制，控制第一级数据输入选择门，并在完成最后一次循环后给出ed信号，指示处理完成，并且对应的数据中线上的数据是有效的，当然也可以使用一个锁存器将该数据存储下来。

**3.3 数据进入方式**

输入构造：hash输入以一个3维bit结构进入，整个结构尺寸为5\*5\*64（1600b），数据则顺序以64bit为单位，放入一个5\*5的二维矩阵中，假设输入矩阵为a每个元素以a（x，y）表示。

数据以8bit（1B）为单位排列，先输入的数据排列在状态矩阵的低位（以8bit位单位，不改变数据的位序），这样从a(0,0)开始排列，直到数据排列完成，然后再进行填充，直到r填充完整。

由于Ethash算法经过设计，填充输入数据长度是固定的，因此填充位置和值也是固定的。

以Hash256为例，采用keccak\_f(1600)(r=1088,c=256)，则输入数据和内部排列如下



可以看到外部输入的第一个数据在内部排列是放在数组序号最小的位置上，以后的数据顺序排列，但是数据的bit顺序不改变，在padding时可以实现10\*1的填充，将内部顺序转换位状态矩阵中数据排列如下：



**4 其他算法**

**4.1 FNV操作**

FNV操作是一种32bit的hash运算，过程为一次32bit乘和一次xor，最后取最低32bit作为输出

FNV\_res = ( (v1\*FNV\_prime)^v2) % 2\*\*32

FPGA设计可以使用一个32bit乘法器和xor操作，需要3个clk完成（32bit乘法2clk，xor使用1clk）



其中，FNV\_prime = 0x01000193。

由于最终结果乘法器输出仅使用低32bit，因此32bit乘法可以简化为3个16bit乘法器和一个16bit加法器，由于仅裁剪低32bit，因此不用考虑加法器的进位。



**4.2 mod操作**

在运算中存在一定数量的mod操作，由于mod除数（b）和被除数（a）都不定，因此采用辗转相减的方法来实现，考虑到数据位数的关系，可以构建一个有限步骤运算完成mod操作。

其基本思想如下：

1）将除数（b）进行左移操作（移出的位记作br），并在右端补0形成b’

2）br及b’与a的大小关系，决定a’的值，若br=0，且a>b’则a’=a-b’，否则a’=a

3）反复运算以上步骤，直到b不再左移，最终结果即位a%b

通过设定移位次数，能够适应一定范围的b运算，假设a为32bit，b为大于26bit的数，则设定移位次数6，则能够完成mod运算，且运算次数不超过7次，实际实现的时候是固定运算周期的，且能够进行pipeline运算。



**4.3 多字节比较器**

多字节比较器用于对多个直接进行直接比较，判断大小或等于，然后输出1bit结果，设计采用并行比较方式进行，可以在clk内完成比较步骤如下：

1. 按直接按字节比较输入值a，b，并获得每个字节的比较结果（假设字节有n个，结果为nbit），结果包括是否等于及大小结果
2. 根据是否等于结果产生一个msk，msk的每位产生规律为，判断该位之前位是否全0，并判断该位是否位1，若是则将msk相应位置1，否则置0
3. 将msk与大小结果相与，判断与结果是否全0，若全0表示判断不满足，否则位满足。



**4.4 多层选择器**

为了增加编译后运行速度，在submux处改用多层选择器，即将原先的1clk完成的32选1，分摊到5clk完成，即对于2^n个数据，采用n层来实现选择，这样每层只需要一层逻辑即可完成，编译后的工作速度可以提高许多。



选择层数n即为Sel的位数，也决定了输入个数（2^n）。

代码书写时建立一个数组Mux来存储输入，输出和中间结果，Mux的大小为2nL+1-2个，初始值放在Mux的0~2^n-1中，最终结果在Mux的最后一个元素中。实现时用公式来索引，以层数n进行循环，设i从0到n-1，每层的选择器个数为2n-1-i个，设循环计数j从0到2n-1-i-1，则每层根据sel(i)进行选择，伪代码如下

for i = 0 to n-1

for j = 0 to 2n-1-i-1

Mux(j+2n+1-2n-i) = (Sel(i) = 0) ? Mux(2j+2n+1-2n-i+1) : Mux(2j+2n+1-2n-i+1+1)

end for

end for

**5 流水hashimoto设计**

系统一个存储器通道带宽为64\*2048Mbps，一次访问数据量为1024bit，若使用一个通道需要约8ns，2个通道则需要5ns，内部模块拟按200MHz时钟工作，则要求对存储器请求为每个时钟请求一次。

前面的设计中，将一次hashimoto过程在一个模块中顺序执行，会导致对存储器的访问间断，假设间断时间为p个clk，则需要使用至少p个模块来填满存储器带宽，当存储器通道数量达到32个（即16组）时，需要的并行模块数量至少为16p个，考虑到一个hashimoto需要的资源（18K个LE和114个DSP），无法产生如此多的模块，

为了满足存储器访问带宽的需求，需要重新设计hashimoto部分，采用流水访问方式，提升数据吞吐率，在存储器访问部分采用流水线方式工作，能够每个clk输出一个访问地址，并接受一个读取结果。

根据数据流架构，proc2部分采用迭代流水线来实现，此外需要将proc1和proc3中的hash拆解开来，以满足流水线贯序输入输出处理要求。

假设Proc2的处理需要a个clk，则流水线可以同时容纳的任务数量理论上为a，要求外部数据输入间隔不大于a个clk，为了缓冲数据，可以建立一个fifo接收外部数据输入，系统设计中a暂定为16（即最多16个任务同时流水处理）

Hash运算主要用于proc1和proc3中，一次hash运算需要执行24次迭代，每次迭代用时3clk，总时间为72clk，为了满足数据输入间隔要求，必须将hash拆分为不少于6段，每段执行4次循环，则流水处理时间为12clk，与proc2数据输入输出需求相当（通常应该更快）。

实际设计中，可能Proc2的处理时间需要18~20个clk，对应的可以通过调节Hash流水级来适应后端流水处理周期，对出口来说，处理速度一定要快于proc2的处理速度，可以用缓冲来实现速率调整。

**流水线任务填充说明**

流水线接收任务后，但任务达到一定数量（MaxTsk），会给出PP\_Valid无效指示，外部则不应当再送入任务。

PP\_Valid指示内部用一个计数器进行当前任务计数，当任务从proc1进入时计数加一，当proc2处理完成后计数减一，之所以这样设计，是因为proc2需要循环多次处理，任务会在proc2中堆积，而后续的proc3采用流水处理不会堵塞任务流，因此以proc2的输出作为任务数减一触发。



最大任务数设置是有讲究的，若该值过小可能导致流水线任务无法填满，最终访问外部存储器时会出现时序上的空隙，降低效率，该值由proc2流水级数与外部存储器访问延迟一同决定

此外，由于任务进入proc1以后，需要一定的时间才能到达proc2，因此需要额外增加几个任务，以补偿proc1的处理空隙，确保流水线能够们负荷运行



其中，Mod运算延迟与Mod级数有关，其位mod级数+1，存储器访问延迟由外部确定，固定延迟（即proc1补偿延迟）由proc1处理周期决定，proc1的hash处理总周期为72，流水化为4级，则每级处理6个循环，一个循环3clk，则总共18clk，对应的固定任务推送数位4个（固定任务数可以这样理解，任务从proc1推入以18clk为周期进入，当一个新任务进入后，需要经过4个18clk周期才能进入proc2，这是由proc1的流水级带来的延迟，这样，为了保证proc2在输出一个任务后，期间不产生处理空隙，需要预备至少4个任务，proc2每输出一个任务后，新的任务进入proc1需要经过4个18clk才能到达proc2，期间proc2会从预备任务中获取一个任务填入处理空隙，当proc2本身是全负荷运行时，每个18clk会结束一个任务，那么在第一个任务结束后，外部随之输入的任务会在4个18clk后再进入，期间又会再输出3个任务，使得流失处理出现连续空隙，因此预备4个任务是必须的，外部也会根据proc2的输出任务填入4个连续的任务，再4个18clk后，外部填入的任务会到达proc2，最为新的备用任务存在。由此可以看到当proc1的流失级变化时推入的固定任务数也会发生变化）

按当前的设计，可以得到如下最大任务数下限计算公式

MaxTsk >= 20+Porc1\_StpNum+ Mod\_Lattice + Mem\_Access\_DL

若Mod\_Lattice为13，Porc1\_StpNum为4（proc1级数），则

MaxTsk >= 37 + Mem\_Access\_DL

注：Mem\_Access\_DL始终大于1，具体由设计决定

注：proc1再级数选择上要遵循proc1一级需用clk数要小于proc2流水线总周期（包括外部mem访问时间），即

n\_proc1\_1stp < 20+Mod\_Lattice + Mem\_Access\_DL

对于当前设计，Mod\_Lattice为13，Mem\_Access\_DL假设为1，则n\_proc1\_1stp应当小于34，根据hash流水设计结果，取4级hash情况下，每级运行周期为18clk，能够满足设计需求，若hash取3级，每级运行周期为24clk，也能满足需求，同时固定任务数可以减少为3个。

20180510

增加ID自动产生器，不再依赖外部ID输入，模块自动初始化ID（个数即序号与内部缓冲大小一致），当推入一个任务同时，产生一个ID，当结束一个任务时，回收一个ID，保证ID整体是平衡的，前提是假设不丢失任务，若存在任务丢失，则ID会减少，最终导致无法开辟新的任务，系统工作出错，此时应当重新初始化ID，即需要系统全局复位

**5.1 proc2设计**

Proc2完成DAG访问地址产生及DAG数据获取等待过程，在该过程中，地址计算可以认为是流水的，在获得上一次的数据后，经过FNV阵列计算，数据选择等操作，即获得下一次需要的地址，值得注意的是，第一次处理，数据来源不是存储器，而是外部输入，这可能会与内部其他过程冲突，为此设计两个FIFO来分别存储外部输入和内部计算结果，通过逻辑进行选择。

考虑到部分数据如s，mix(n-1)等需要跨周期使用，引入存储器来进行存储数据，并通过地址索引方式来重现数据，为了设计方便，认为nonceID即为存储器的地址，要求该值不大于64（即存储器大小，实际应用中不超过32），信号流结构如下



注：FIFO采用Throughout方式，即读信号作为ack使用，输出端口上的数据即为当前需要传输的数据。

注：logic1产生mux的选择信号，但i为0时选择s0输出，否则选择mix(n-1)输出，该逻辑为1clk

注：logic2产生mux选择信号及fifo读写信号，该逻辑不延迟，逻辑关系为当mem\_valid有效，优先传输mix数据，不能产生读信号，当mem\_valid无效，不产生读信号（该信号经过延迟后，作为后级控制信号）。

注：存储器使用双口ram即可，为了获得s的重复访问，开设两个存储器，存储地址和内容相同，但是读取不同，方便设计，使用altera芯片，10K块存储器，预计需要13+13+26=52个，FIFO存储器预计需要不多于10个（此外proc3还需要缓存输出约需要不少于36个）

注：资源预估，LE资源约10K，ram资源共90个

注：更改，去掉对Si的存储，该存储放到proc3中实现以节省存储器，该存储器的写入可以直接用proc1的输出执行，按nID地址写入，理论上在一个nID完成前，后续输入的nID是不会重复的，因此可以将proc1的输出保留到proc2的输出以后，并按nID进行访问



**5.2 Hash设计**

Proc1/3中主要是Hash部分，其他的都已经流水线化，因此只要将Hash流水线化即可，proc1/3处理中不再使用FIFO进行中间过程缓存，进入的数据经过处理后直接推到后级输出，处理周期为Hash流水级处理周期，对应的输入数据也应当以该周期进行，设计中会额外给出控制信号和计数器来表示当前流水线工作状态（包括是否允许数据输入）

Hash处理过程本身就是一个流水过程，但是需要多次循环，已经设计的处理核中已经考虑了循环次数可配置，因此只需要重复处理核即可。



一次keccak计算需要使用3clk，若采用4级流水线，则每个核需要运行24/4=6次循环，共18clk，实际设计是支持调整的。

注：altera编译后需要资源使用及流水周期表：

流水线级数 LE 流水周期

1 6.2k 72clk

2 11.3k 36clk

316.5k 24clk

4 21.6k 18clk

6 31.9k 12clk

8 42.2k 9clk

12 62.8k 6clk

24 124.5k 3clk

**5.3 proc1设计**

Proc1使用一个fifo来缓存外部到来的nonce，内部使用一个流程控制器，判断Hash3是否可以接受任务，若可以则从FIFO中取出nonce，合并head进行hash操作，并将结果输出到proc2中（以hash的ed信号作为当前数据结束信号即可）

另设计有一个fifo用于存储nID，为了保证时序同步，使用寄存器锁存输出信号一次



**5.4 proc3设计**

Proc3接收来自proc2的输出，并执行一次FNV压缩和一次Hash，需要FIFO来同步输出的cMix数据以及缓冲hash的数据。



注：更改hash，输出preEd，即在Ed的前一个时刻输出一个信号，这样可以减少一些D触发器延迟，该更改影响proc1/3

**5.5 DAG设计**

**5.5.1 Proc2**

考察DAG结构，其proc2过程与hashimoto的proc2过程基本一致



考虑与hashimoto合用proc2结构，则对hasimoto的proc2主要修改包括：

1. S0输入增加FIFO存储，DAG的i输入也需要存储，通过功能进行选择
2. ~~S0/i从Mem输入时进行选择，有1clk延迟，去掉原有的1D延迟~~（不要了，因为对外地址同意编制，接口只有一个就可以了）
3. Logic3计数边界可选，DAG为256次，hashimoto为64次
4. Mux输出后，缓存FIFO增加对S0/i的存储接口，输出同样增加
5. Info数据结构中，增加S0/i项
6. AddrOffset增加cache值，内部进行选择
7. Mix宽度两种处理不同，但是都按128bit进行，仅仅在最后输出链接时选择，submux时是会有区别的
8. Mod输入b根据模式选择
9. Submux的选择根据模式区别，DAG为6位，hashimoto为7位
10. 输入增加一个模式选择

Proc1增加一个前端输入，后端仍然使用hash512（不同模式点数不同）

Proc3为2个，不同模式不同处理



**5.5.2 Proc3**

ADG的Proc3与hashimoto的proc3仅共用hash模块，其余部分单独实现，控制器合并，通过Mod\_Sel来选择需要增加的部分包括Hash输入选择，hash的结束信号路由（路由仅一个逻辑，可以不用同步）。



主要更改包括：

1. 输入Mix和St增加直接进入FIFO的选择控制
2. RAM也需要存储j，与原有的S共用低4B，选择器无时钟同步
3. 进入Hash数据进行选择，对应的Hash的St需要1clk延迟
4. 增加一个FIFO，单独存储从RAM出来的j并输出
5. 所有控制信号需要增加一个选择控制，用于切换

20180507

额外增加一个保存nonce的ram，由St输入直接将nonce写入，然后在proc3结束的时候在取出，随着结果以通输出

**5.5.3 Proc1**

ADG的Proc1相对hashimoto来说增加了获取Cache数值的过程，需要增加一个FIFO以同步外部访问过程



整个模块分为3个相对独立的部分，包括原有Hashimoto处理部分，cache地址产生部分和DAG处理部分，三部分基本独立工作，共用hash模块及输出

**6 存储器访问**

存储器访问需求可以定义为多个访问请求（来自多条流水线）共同访问同多个存储器，由于访问地址的产生是随机的，因此访问可能存在冲突，在访问前需要根据地址进行重新分配（多个存储器按地址分块）。当存储器输出访问结果后，需要将结果重新推回访问请求放，由于前面已经执行的重分配，因此后面也需要重新分配到对应的出口，此外存储器有刷新时间等一系列指令握手协议构成，其内部时序不可控制，会带来访问时序上的不确定，其也会严重的造成重分配时刻的冲突，为此需要设计一套机制，有效的降低冲突引入的访问效率下降。

**6.1 策略一**

考虑输入情况，入口请求来自于hashimoto流水线，其工作于流水线方式，可以达到每个clk会输出一个请求，请求中的地址是用于访问存储器的地址，存储器被划分为n块（n=16/32），那么理想情况下，每条流水线的输出可以立刻访问一个对应的存储快，但是由于访问地址的产生方式是随机的，因此有一定的概率产生冲突，即同一时刻有多个请求要访问同一个地址块，解决冲突的方式即将同一块访问请求进行排队，并尽量减小由于冲突引起的访问延迟。

每条流水线的访问请求被推入一个FIFO中，在出口端读取FIFO中的数据，一次读取一个。

利用反馈方式将冲突的数据反推入备用FIFO中，然后根据策略来选取一个数据读出。

整个设计过程依然遵循流水线方式进行。

为了提高效率，将地址分组处理，可以达到较好的效果，鉴于外部存储器有32个，因此实际效果会更好，利用matlab仿真，得到如下统计结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 输出通道数 | 输入通道数 | 地址分块 | 分块后通道数 | 效率 |
| 16 | 16 | 1 | 16 | 64.58% |
| 16 | 16 | 2 | 8 | 87.2% |
| 16 | 16 | 4 | 4 | 99.78% |
| 32 | 16 | 1 | 32 | 79.67% |
| 32 | 16 | 2 | 16 | 100% |
| 32 | 16 | 4 | 8 | 100% |



实际操作中需要采用一些策略用于提高效率：

1. 反馈FIFO数据取出不是马上取出，而是缓存一定的数据个数后取出（当两个FIFO都有数的时候），可以设定固定值（仿真中也采用了随机值和与输入FIFO中值比较大小的情况，效果差不多）
2. 在B\_M中判断选择那个数据，不能一直从第一个开始，否则后面的数被选中的机会就小了，导致输入FIFO数据堆积，可以采用轮询的方式（即每次开始查找的位置都不同，可以采用顺序移位的方式进行），让所有的通道机会均等

**6.2 策略二**

**6.2.1 设计架构**

该策略下，使用存储器进行工作，输入地址存储在存储器中，并且将存储器中全部的地址请求都参与选择，找出不冲突的请求，并进行申请，该策略下，需要对地址进行附加一个内部地址，根据地址存入存储器响应的位置，并标记该位置上是否存在数据，在选择时便利存储器中有内容的位置，选择一个不冲突的地址进行访问



该策略的思路是，输入按唯一地址Addr进行随机位置存储，输出则全局判断是否存在与前面模块不冲突的值，然后输出缓存到FIFO中，即将输入流重新排序为相互之间不冲突的输出流，然后利用FIFO进行对对齐后就可以直接输出了。

该设计的代价是需要额外的存储空间，并且选择过程中需要大量的逻辑展开，当RAM深度较深时，逻辑使用量及其大，为了保证时序需要多级打拍（例如RAM深度为128，则进行ADDR和ch选择的时候要2^=128选1，逻辑实现较复杂，可以拆分为7级流水线实现），从而导致一个模块的输出延迟较大，对应的第一个缓存FIFO的深度也要很大。假设延迟为n\_dl，模块有p个，则第一个缓存深度至少为p×n\_dl。

减少延迟的一个策略是降低RAM的深度，初步考虑是当RAM中存在足够的值时，每次hit中的概率就会增加，若有p个模块，则RAM中堆积的数不需要超过2p，实际需要通过仿真来确定

**6.2.2 数学分析**

该策略下无堵塞，可以这样来建模，假设由n个篮子，每个篮子中有m个球，每个球上随机标注序号（范围为1~n），依次从n个篮子中取球，要求每个篮子中取出的球上的序号与前面所有篮子中已经取出球的序号不同，问第i个篮子中至少有多少个球时，可以大于一定概率p找到符合要求的球。

1）事件a1：第i个篮子中的1个球与前面的1个球数值不同

P(a1) = 1-1/n

2）事件ax：第i个篮子中的1个球与前面i-1个球数值不同

P(ax) = ∏P(a1) = (1-1/n)^(i-1)

3）事件A：第i个篮子中m个球中至少有一个与前面i-1个球数值不同

P(A) = 1-P(Ab)

即等效于m个球都存在与前面i-1个球的数值相同的情况

P(Ab) = (1-P(ax))^m = [1-(1-1/n)^(i-1)]^m

则问题的事件A的概率为

P(A) = 1-[1-(1-1/n)^(i-1)]^m

对于要求的概率小于p，则有

P(A) = 1-[1-(1-1/n)^(i-1)]^m < p

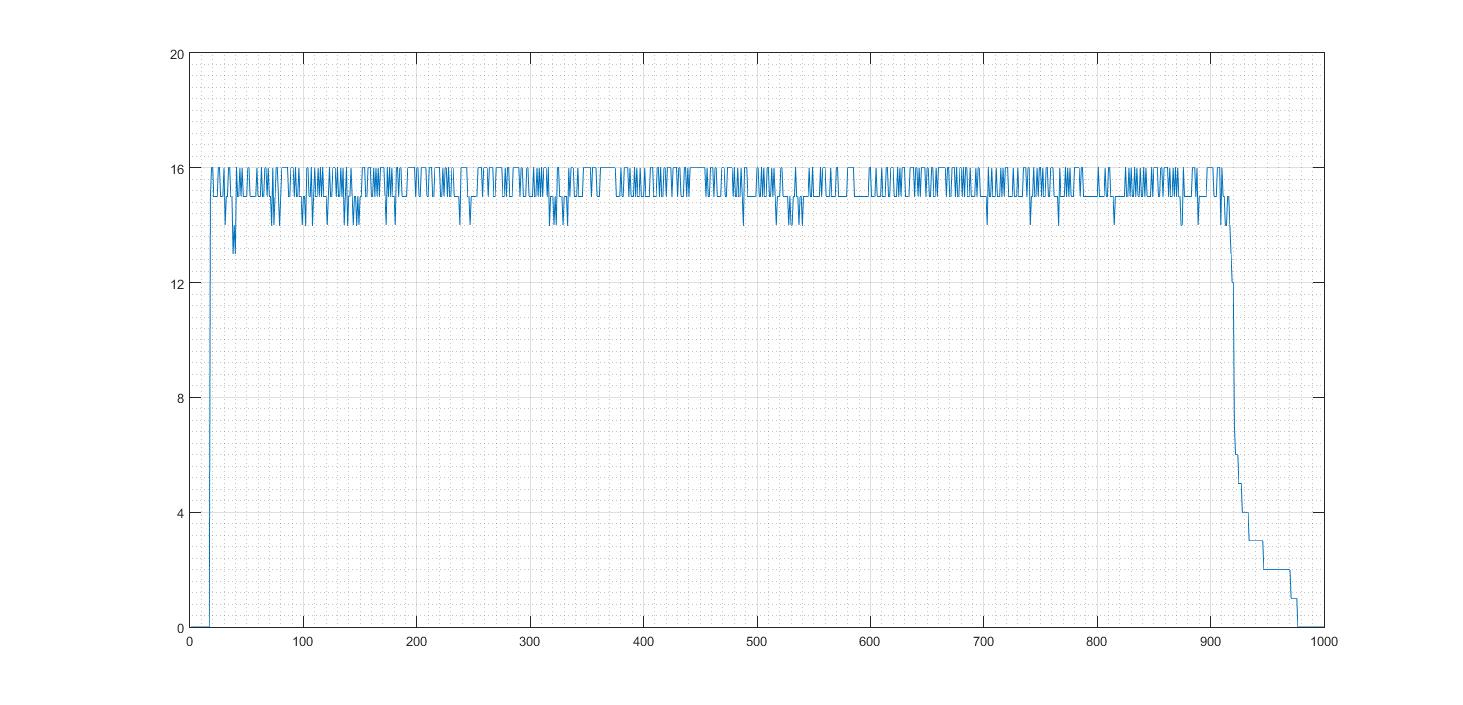
两边取对数有

m > log10(1-p) / log10(1-(1-1/n)^(i-1))

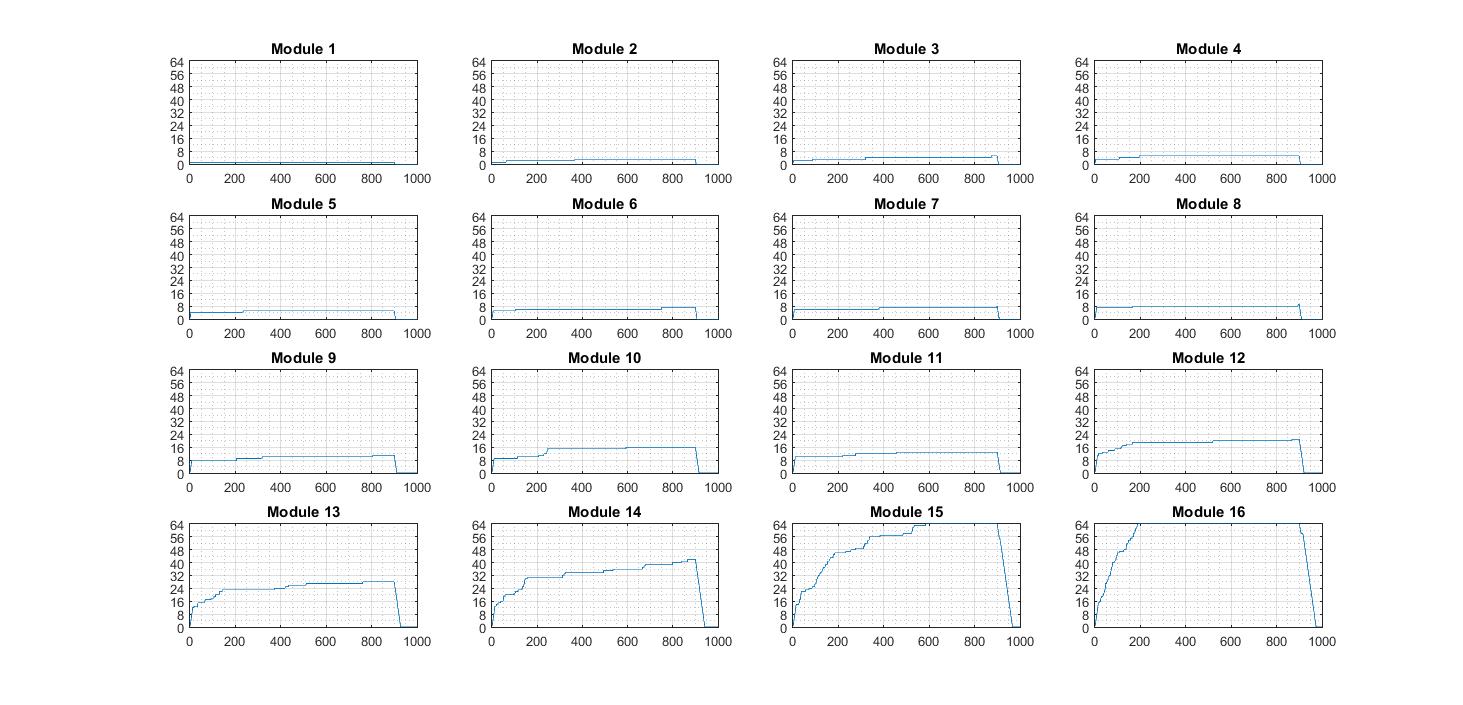
假设p为99.999%，n为16，i从1~16，则有

| 1-p | i | | | | | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 1.00E-02 | 0 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 9 | 9 | 10 |
| 1.00E-03 | 0 | 3 | 4 | 4 | 5 | 6 | 7 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1.00E-04 | 0 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 17 | 18 | 20 |
| 1.00E-05 | 0 | 5 | 6 | 7 | 8 | 9 | 11 | 12 | 13 | 15 | 16 | 18 | 19 | 21 | 23 | 25 |
| 1.00E-06 | 0 | 5 | 7 | 8 | 10 | 11 | 13 | 14 | 16 | 17 | 19 | 21 | 23 | 25 | 27 | 29 |
| 1.00E-07 | 0 | 6 | 8 | 10 | 11 | 13 | 15 | 16 | 18 | 20 | 22 | 24 | 27 | 29 | 32 | 34 |
| 1.00E-08 | 0 | 7 | 9 | 11 | 13 | 15 | 17 | 19 | 21 | 23 | 25 | 28 | 30 | 33 | 36 | 39 |
| 1.00E-09 | 0 | 8 | 10 | 12 | 14 | 17 | 19 | 21 | 23 | 26 | 28 | 31 | 34 | 37 | 40 | 44 |
| 1.00E-10 | 0 | 9 | 11 | 14 | 16 | 18 | 21 | 23 | 26 | 29 | 31 | 35 | 38 | 41 | 45 | 49 |
| 1E-11 | 0 | 10 | 12 | 15 | 18 | 20 | 23 | 26 | 28 | 31 | 35 | 38 | 41 | 45 | 49 | 54 |
| 1E-12 | 0 | 10 | 14 | 16 | 19 | 22 | 25 | 28 | 31 | 34 | 38 | 41 | 45 | 49 | 54 | 58 |
| 1E-13 | 0 | 11 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 37 | 41 | 45 | 49 | 53 | 58 | 63 |
| 1E-14 | 0 | 12 | 16 | 19 | 22 | 26 | 29 | 32 | 36 | 40 | 44 | 48 | 53 | 57 | 63 | 68 |
| 1E-15 | 0 | 13 | 17 | 20 | 24 | 27 | 31 | 35 | 39 | 43 | 47 | 52 | 56 | 62 | 67 | 73 |
| 1E-16 | 0 | 14 | 18 | 22 | 25 | 29 | 33 | 37 | 41 | 45 | 50 | 55 | 60 | 66 | 71 | 78 |
| 1E-17 | 0 | 15 | 19 | 23 | 27 | 31 | 35 | 39 | 44 | 48 | 53 | 58 | 64 | 70 | 76 | 82 |
| 1E-18 | 0 | 15 | 20 | 24 | 28 | 33 | 37 | 41 | 46 | 51 | 56 | 62 | 68 | 74 | 80 | 87 |
| 1E-19 | 0 | 16 | 21 | 26 | 30 | 34 | 39 | 44 | 49 | 54 | 59 | 65 | 71 | 78 | 85 | 92 |
| 1E-20 | 0 | 17 | 22 | 27 | 32 | 36 | 41 | 46 | 51 | 57 | 62 | 69 | 75 | 82 | 89 | 97 |

进行1e5次随机数产生实验，仿真结果与设想一致，可以实现输入输出无堵塞，存储器容量需求及输出情况如下，可以看到基本满足设计情况



当前时刻数据输出个数



RAM残留数据个数

通过仿真可以看到，最后一级存储器仍然有可能增大，这个是由于最后一级存储器每次可选的余地都相对较小，为了改善这种情况，需要启动的时候每个存储器一起启动，但是这样就无法实现数据对齐，这个问题需要考虑一下。

以上设计是错的

**6.2.3 流水运算先验补偿**

在实现过程中存在一个问题，Ram的flag需要在完成选择运算后更改，但是选择运算需要较多步骤才能确定更改哪个flag，这会导致控制无法流水工作，为了解决这个问题，要求f存在ag更改在下一个clk就实现，即控制逻辑无延迟，这个实现可能比较困难

目前考虑的方法如下：在sel运算结束时，清除寄生在Ram下的flag，从计算出可用的值到清除flag之间的结果，都要或在一起，用于修正输入结果，这样就可以保证flag的同步，得到正确的计算结果，同时由于系统设计中确保了一个flag置位之后，在清除前不会再次置位，这个是由任务序号发生器（一个FIFO）保证的。



实际设计中，清flag标志可以稍微提早一点，以减小对or操作的压力，这不影响Addr的返还和后续的操作，因为后续需要的参数都会保留在处理流水线中，Ram中存在的数也不会被清除，而且Addr没有返还前，是不会对该位置的数进行新的操作的（新的写入操作）

该设计可以这样理解：假设输入a序列受到输出结果即新输入的影响，a(i)被推入流水线进行处理，假设经过n个clk后出结果b(i)，然后再经过m个clk后输出修正输入a序列，而此时未被影响的a(i+1)值已经被推送入流水线中进行计算，并且在第n+1时刻会输出结果，此时为了修正输出结果，则需要将a(i)的结果b(i)参与到a(i+1)的处理中，从而产生正确的b(i+1)值，对于a(i+2)的运算，则需要使用到b(i)和b(i+1)值，依次类推，直到a(i+n-1)，需要用到b(i)~b(i+n-2)值，考虑到影响输入需要再经过m个clk，则从a(i+n)到a（i+n+m-1）的运算也需要同样的修正。

假设a为一个16bit序列，流水处理的操作为先将a(i)寄存1clk，然后找出第一个1（1clk），反馈修正操作为将a(i)中该位清零，操作延迟3clk，输入操作位顺序设置a(i)中的1位为1，则系统处理架构及时序如下，时序图中着色部分表示了数据处理的来源关系





**6.2.4 FPGA实现**

策略二的一个模块的实现结构图如下